ASSURED ANDROID EXECUTION ENVIRONMENTS

THESIS

Brandon Froberg, Capt, USAF

AFIT-ENG-MS-18-M-027

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government.

AFIT-ENG-MS-18-M-027

ASSURED ANDROID EXECUTION ENVIRONMENTS

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

Brandon Froberg, B.S.C.E.N., M.B.A.
Capt, USAF

March 2018

AFIT-ENG-MS-18-M-027

ASSURED ANDROID EXECUTION ENVIRONMENTS

THESIS

Brandon Froberg, B.S.C.E.N., M.B.A.
Capt, USAF

Committee Membership:

Dr. Laurence D. Merkle
Chair

Dr. Scott Graham
Member

Dr. Timothy Lacey
Member

AFIT-ENG-MS-18-M-027

Abstract

Current cybersecurity best practices, techniques, tactics and procedures are insufficient to ensure the protection of Android systems. Software tools leveraging formal methods use mathematical means to assure both a design and implementation for a system and these methods can be used to provide security assurances. The goal of this research is to determine methods of assuring isolation when executing Android software in a contained environment. Specifically, this research demonstrates security properties relevant to Android software containers can be formally captured and validated, and that an implementation can be formally verified to satisfy a corresponding specification.

A three-stage methodology called "The Formal Verification Cycle" is presented. This cycle focuses on the iteration over a set of security properties to validate each within a specification and their verification within a software implementation. A security property can be validated when its functional language prototype (e.g. a Haskell coded version of the property) is converted and processed by a formal method (e.g. a theorem proof assistant). This validation of the property enables the definition of the property in a software specification, which can be implemented separately in an imperative programming language (e.g. the Go programming language). Once the implementation is complete another formal method can be used (e.g. symbolic execution) to verify the imperative implementation satisfies the validated specification. Successful completion of this cycle shows a given implementation is equivalent to a functional language prototype, and this cycle assures a specification for the original desired security properties was properly implemented. This research shows an application of this cycle to develop Assured Android Execution Environments.

Table of Contents

Page

List of Figures

List of Tables

ASSURED ANDROID EXECUTION ENVIRONMENTS

I. Introduction

I.1 Current Issues with Android Security

In 2017 it was reported that Android comprises two thirds of the smart phone market. In the same year the operating system (OS) hit a new milestone: two billion monthly active users [1] [2]. The market saturation and active number of users indicates that approximately one in four people in the world use an Android device.

Unfortunately, many of the types of threats to desktops and servers also apply to mobile devices. A way to quantify these threats on Android is to consider MITRE Corporation's Common Vulnerabilities and Exposures (CVEs). Each CVE describes a vulnerability, usually in significant detail, as well as its vector of attack and the impact on a software product. One source, CVEDetails.com, has analyzed and consolidated CVE information into 13 categories of vulnerabilities [3]. These categories have seven major classes of threats to Android. The data from the website from 2009 through 2017 shows there are 1533 CVEs, and the breakout is the threats are seen in Figure 1. This information is independently verifiable with analysis from the National Vulnerabilities Database (NVD) using scripts found in Appendix A.2.

Malicious actors generally seek new methods and resources for their attacks, and those targeting the Android operating system are no exception. All categories of CVEs are growing, and some threat categories (e.g., code execution, overflow, gain information, gain privileges) have seen over 300% growth within the past few years.

Evaluation of the CVEs indicates that 83% of the vulnerabilities on Android

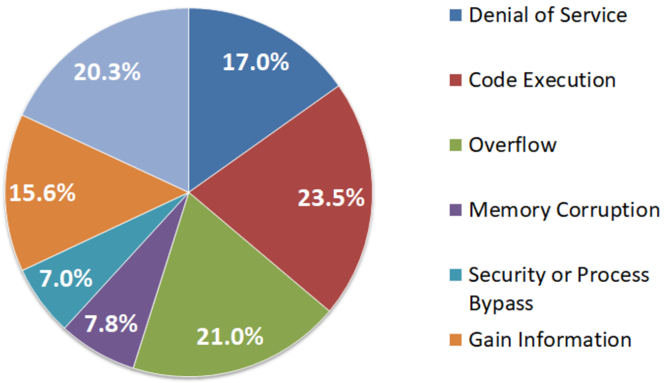


Figure 1. Android CVEs by Vulnerability 2009-2017

impact the proper execution and protection of information (i.e., the vulnerabilities described by the CVEs, except for Denial of Services, affect applications and their data). The top three CVE classes in Figure 1 can be interpreted as mechanisms used to execute code or gain information without OS permission.

One way to manage such threats and vectors is virtualize and isolate applications, because a contained environment prevents undesired access as well as unintended access (e.g., overflow access). Although not the primary goal one goal of the Android Security Sandbox is application virtualization and isolation. However, a concern with the sandbox is that its design and implementation is open to the vulnerabilities described by the CVEs. At this time there is no public documentation that shows validation of the sandbox's specification or verification of its implementation. Therefore, a new containment solution is needed to assure a sandbox-like design fulfills security claims.

I.2 Problem Definition Development

The current and future threats to Android are such that new and enhanced security measures are needed. One technique to reduce cyberattack surfaces, by validating a specification and verifying an implementation, relies on formal methods. The current literature does not claim the use of formal method tools and techniques in the development of Android or its applications. Android lacks formally verified solutions specifically that has assured containment of (1) malicious application execution on a trusted device, and (2) a trusted app on a malicious device. Therefore, this research explores the application of formal methods to assure Android containment. With a primary focus on malicious software applications.

I.3 Research Questions and Hypothesis

As it stands there is no established or *de facto* standard set of methods for the formal assurance of software containment. Also, there is no standard set of security properties for software containment. Hence, this research seeks to determine what security properties are required for Android containment that can be assured through formal methods, and to establish representative methods that can be used to validate security properties within a specification and verify the subsequent implementation of a software product. In particular, this research is seeking answers to the questions:

- "Can an assured software container system be developed for Android?"
- "What would be the methods to accomplish this task?"

To answer these questions several tasks need to occur including:

- Determine if any formally verified products can be ported to Android
- Develop formal methods to support a verified execution environment on Android
- Prototype an implementation of an validated specification

By completing these tasks, this research demonstrates that containment security properties can be identified, validated, and captured in a specification and implemented in a verified Android software container. The following sections detail the planned approach, outline research contributions, and provide an overview to the thesis.

I.4 Approach

For an assured Android containment product to be realized a set of security properties must be identified. Additionally, these security properties must be assurable with formal methods. Specifically once the properties are defined, a specification of each property must be validated and its implementation in a programming language must be verified. Thus, three stages of research are presented to illustrate the proposed formal approach. The first stage consolidates a set of security properties that can be validated with formal methods. The second and third stages demonstrate formal processes for the validation of a given property in a specification and its verification in an implementation. Specifics of these stages are detailed in Chapter 3, and related findings are presented in Chapter 4.

I.5 Research Contributions

This research provides three specific contributions:

- A formalized set of security properties for Android containment specifications
- A proposed Android Framework for formal assurance
- Proposed techniques of formally Android specifications and implementations

I.6 Thesis Overview

The second chapter discusses the technical background and related research pertaining to both formal methods and Android containment. Chapter 3 presents Assured Android Execution Environments, which is a framework that encapsulates the proposed security properties and the Formal Verification Cycle. Details relating to the three stage approach used in this research are outlined emphasizing the tools used to test the proposed framework. Chapter 4 details the results and findings, and the final chapter presents the research conclusions and recommendations for future work.

II. Background and Related Research & Technology

This chapter examines the four primary technology areas needed for this thesis (i.e., formal methods, containerization, Android, and existing security models and frameworks), and concludes with a discussion on related research. Due to the complexity of some of the topics, both introductory discussions and specific technology implementations are provided. This information is intended to serve as both an introduction and reference guide for readers and future researchers alike. Additionally, some of this documentation familiarizes key technologies, because of their use for modeling the approach, design, and implementation methods for Android container assurance.

II.1 Formal Methods

General Background

Before showing examples of what formal methods, or how they are used in this research, some definitions are needed. According to Schmeelk, formal methods "are a set of mathematical representations of a system which can be verified," and formal assurance "is the attempt to use formal methods to validate design requirements or specifications" [4]. Additional terms needing definition are validation and verification, and with respect to this research these two terms will adhere to the definitions provided by Tran via the IEEE Standard Glossary [5]. Validation is an evaluation to determine if a system satisfies specified requirements, and it can happen during or at the end of a development process. Verification is a process to evaluate developed products of a system satisfying the conditions imposed at the start of that design phase. A final topic to consider is Software Engineering, which is defined by Pressman as the establishment and use of sound engineering principles to obtain software that is reliable and efficient [6].

This research presents a working definition for formal assurance as: the use of formal methods to (1) validate the original requirements of a specification and/or (2) verify developed products satisfying a (validated) specification. Formal assurance shows the correctness of a design when leveraging both validation and verification, and the specific formal methods techniques used can vary for either validation or verification. Also, shorthand terms for formal assurance (e.g., assurance, assured and assurable) are based on the working definition in this thesis.

Schmeelk states that correctness of a program can be shown by forms of approximation, and the three most common forms are abstract interpretation, theorem proving, and model checking [4]. Hence, if these forms of approximation are used successfully with proper validation and verification, then certain assurances are proven for a given specification and its implementation.

This showcases how formal methods, and the assurances thereof, differ from classically used approach of tests. In general, formal method proofs differ from unit and regression testing. Testing does not demonstrate full, proper operations of a system other than for the specific cases under test [7]. In particular, testing shows the result for a specific input, while a formal proof shows a general result for an entire class of inputs [8]. These general results are possible for formal methods because they model a system's operations with a rigorous mathematical representation [4]. Thus, in attempts to secure a system or ensure security property design, then the benefits of formal methods greatly exceed those of simple unit tests.

An Example Approach in Formal Validation: Functional Programming Language Prototypes

To assure security properties via validation, a proof engineer needs to take care in their claims with respect to the semantics of a program. A common practice for software design is the construction of a prototype. If an imperative programming language is used, then the specification will a state-based implementation. Such an implementation has inputs that modify its state, and the implementation may have non-deterministic execution (i.e., the behavior of such a program depends on both the current inputs and the state resulting from execution on past inputs). As such, to ensure that mathematical analysis and program determinism is as practical as possible, a functional programming language should be considered over an imperative programing language implementation.

Functional programming languages process inputs to produce an output. There are two major implementations of functional programming languages: pure and impure languages. Pure languages are seemingly the ideal case of computer execution of an algorithm (i.e., a step-wise and defined set of logic or math is applied to inputs that result in outputs). However, actual software programs typically have "effects," which can be actions such as: a global state, exception handling, non-deterministic type outputs, assignments, and continuations [9]. This is where impure functional programming languages have been created, since these languages allow for effects to be implemented on top of the functional programming language features [9]. Typically, impure functional programming languages are realized with the addition of "monads" (i.e., monads allow functional languages to include and account for program effects) [10]. For example, a program that reaches a halting state (e.g., an exception handling of a calculation that was produced from a division by zero or the resulted as "not a number") would result from monad exception handling. Monads can also give a uniform framework for program development [10].

In order to leverage monads for larger claims, and for prototyping with functional languages, a team from the National ICT Australia (NICTA) Limited showed a method to validate their monadic design with a proof script that was validated by a larger proof assistant tool [11]. The team leveraged a set of mathematical techniques known as Hoare Logic. Applied Hoare Logic is used to evaluate and validate Hoare Triples, which are comprised of an assertion of the state of the code before execution (i.e., the precondition), the code fragment (i.e., the algorithm to manipulate data), and an assertion of the state after execution of the code fragment (i.e., the post-condition) [12]. The Hoare Triple enables a method to define code that is modeled and bounded to assure the code fragment's execution. They are the foundation for the calculus and rules of operation for specific actions or code assignments that are proven valid or invalid [12]. The features of the seL4 team's monad, were validated with Hoare Logic, to validate the team's claims regarding the security of their system under test [11]. Formal methods usually try to prove a base case and build upon proof of work. The NICTA team showed that their monad was correct and then used it for functional language prototypes to enable security properties [11].

Survey of Formal Verification Tools & Projects

Formal Verification Projects and Tools

The literature described a significant number of formal verification projects. Chong, et al., provided a detailed list and examples reviewing numerous projects, and the most relevant of these projects and tools are shown in Table 1 [13].

Table 2 lists a representative range of tools used in formal verification such as proof assistants and model checkers. Many tools were found based off the work by Chong, et al., but the work of Armstrong, et al., greatly expanded the total identified

Table 1. **A Selected List of Formal Verification Projects and Applications**

Project Name	Features
seL4	A formally verified L4 microkernel that runs on ARMv6/v7 and x86 processors
CertiKOS	Certified concurrent, general purpose operating kernels
ExpressOS	A mobile OS enabling high assurance applications with formal methods proven security invariants
MinVisor	A Type-I x86 hypervisor with proven protection properties at the assembly level using ACL2
Rocksalt	Software-based fault isolation, as used in Google Chrome's Native Client
Jitk	In-kernel interpreters; native instruction execution of compiled user-space policies
FSCQ	First file system with a machine-checkable proof which its proven specification includes crashes
XMHF	A modular, high performance hypervisor framework with automated verification
Verve	An Operating system verified with typed assembly language and Hoare logic

formal verification tools [14]. The tools included cover hardware, software, and general system modeling.

Two specific tools have been foundational in this research: Isabelle and KLEE.

Isabelle

The NICTA effort, to leveraging Hoare Logic to prove and validate monad design, was implemented with Isar. Isar is a modeling and "proving" language that allows for creation of theorems understandable by both humans and computers (i.e., the theorem that is created can be read directly from Isar source files and be understood) [15]. Such a language allows the creation of lemmas and theorems for a system as a set of theory files (i.e., mathematical representations can be made of designed logic and assumptions). Constructed theories can then be mathematically proven to be true or false by the proof-assistant environment Isabelle/HOL, commonly referred to as Isabelle. Describing these theories in Isar allows for higher-order logic to be used as a logical calculus for validation efforts [15]. The NICTA team used Isar to capture and represent their monads under test, saved their theory files in an Isar format, and used Isabelle to validate their designs. This technique is representative of the theorem proving form approximation referred to by Schmeelk.

Table 2. A Selected List of Formal Verification Tools and Methods

Application Name	Platform	Features
Alloy	Java	A language and tool that evaluates structures by reduction to SAT
BLAST	Linux	A Model Checking Tool for C Programs
CacheAudit	OCaml on Linux and Mac	Static analysis tool for cache side-channels
Caisson	VHDL/Hardware Design	Its a provably secure HDL that is converted to Verilog
CMBC	Windows, Linux, Mac	A C/C++ Bounded Model Checker
CertiCrypt	The Coq Proof Assistant	A cryptographic proof assistant
CertiPriv	The Coq Proof Assistant	A reasoning framework for differential privacy
Code-Pointer Integrity	LLVM	It assures program code pointer integrity
Cryptol	Windows, Linux, Mac	A programming language for cryptographic algorithms
CryptoVerif	OCaml	Automated security protocol assurance
DJoin	N/A	Differential reasoning to process distributed database queries
DFuzz	OCaml	A differential privacy type checker
EasyCrypt	OCaml	A cryptographic proof assistant
Facebook Infer	Linux and Mac	A Static analysis tool for C, C++, Java and Android
FDR/FDR2	Windows, Linux, Mac	A model checker based on communicating sequential processes
Frama-C	Windows, Linux, Mac, FreeBSD	Program analyzers for C
GUPT	N/A	A tool for guarantying differentially private systems
GLIFT	N/A	Tools for analysis, statically verification, and controlling information-flows
Incisive	N/A	Hardware, at a gate level, safety policy simulator
Isabelle/HOL	Windows, Linux, Mac	A generic proof assistant using Isar proof script
JasperGold	Incisive	Functional checker and debugger for register-transfer level hardware
Java Pathfinder	Java	An execution environment for bytecode verification
KCoFI	N/A	A system to ensure control-flow integrity protections
KCoFI	N/A	A system to ensure control-flow integrity protections
KLEE	N/A	Analysis framework for LLVM-IR to symbolically execute C/C++ Code
NuSMV (and SMV)	Windows, Linux, Mac	An open architecture for model checking
Pinq	Windows	Reasoning tools for differential privacy properties
Questa Formal Verification	N/A	Hardware simulator to determine behaviors and error states
rF*	F*	A cryptographic security proof verification system based on refinement types
Sapper	N/A	A hardware description language for security-critical designs
SC-Sniffer	N/A	A side-channel leak analysis tool
SecVerilog	Linux	A hardware description for information flow reasoning
Simulink Design Verifier	Simulink	Dynamic and static analysis for model error detection
SPARk	Ada	An Ada based language and set of tools to ensure software integrity
Spin	Promela	A promela interpreter that generates C code for model checking
SVA	LLVM	A virtual operating system for analysis and modeling of software
UPPAAL	Windows, Linux, Mac	A real-time system modeler via timed automata
VDM	N/A	Vienna Development Method: a set of methods and tools for formal reasoning

KLEE

According to Schmeelk, common methods of abstract interpretation include: decision tables, symbolic execution, static and dynamic code analysis, and border-line informal methods [4]. The method most relevant to this research is symbolic execution. According to King, symbolic execution focuses on supplying symbols instead of the normal inputs for a program, and the execution proceeds normally with symbolic formulas over the input symbols [16]. The advantage of symbolic execution for a program is a set of inputs is symbolically executed, and these inputs may be equivalent to a large number of normal test cases [16]. Furthermore, the results of symbolic execution can be checked for correctness either formally or informally [16]. One such approach is to use a language independent intermediate code representation. This provides a means to determine the states and pathways of execution independent of the original source code.

One intermediate representation is LLVM Intermediate Representation (LLVM-IR), which is widely used for formal analysis [17]. One tool based on LLVM-IR is KLEE, which uses symbolic execution to generate test inputs. KLEE includes a symbolic library into a C program under test, and then compiles the result to an LLVM-IR format. KLEE then uses built-in heuristics to test the symbolic space and determine program semantics. Tools such as KLEE allow proof engineers to test implementations of their validated theories, thereby providing a holistic approach to testing.

Using seL4 as a Model System and Approach

The seL4 microkernel, created by the Software Systems Research Group at NICTA, now Trustworthy Systems at Data61, has a validated design and verified implementation [18][19]. The group has since released tools and published techniques to allow continued development of seL4 and its applications. The development process is focused on the use of automated tools in the formal analysis and kernel prototyping of an L4 microkernel. The design, proposed by Jochen Liedtke, embodies the functional design of microkernels: the kernel provides a set of general mechanisms while user-mode servers implement the actual OS services [20]. A new framework was created on top of the verified kernel to enable assured application development. This framework extends the secure base of the kernel to allow developers to build upon the proven work and designs of seL4. Such a capability can drastically reduce the effort needed to assure new features and designs.

An Observation of a Formal Methods Process

Klein, et al., showed that assurance is achievable with formal machine-checked verification, such that mathematical proof and implementation is consistent with its specification and free from programmer-induced implementation defects [19]. The seL4 team used a refinement proof to establish a correspondence between high-level/abstract and a low-level/concrete representation of a system. The correspondence proof ensures logic (specifically Hoare logic) properties of the abstract model also hold for the low-level representation [19]. This implies that if a security property is proved (in Hoare logic) about the abstract model, then refinement guarantees the same property holds for the implementation source code [19]. As the seL4 team pointed out the source code would still need to be validated to ensure it meets the code representation [19]. Hence, a given design for an abstract theory could be input to an automated tool to prove its properties are valid, which allows for guarantees regarding the source code representation/implementation. It is observed that desired theoretical properties can have their design validated leading to verification of an implemented specification. Modeling these techniques allows for a standardized approach to formal methods.

Leveraging Automated Tools in Design Validation

One may assume that in order to solve proofs the only method is to directly solve and leverage a proof assistant's primary language/scripting language. When delving into the methodology of the seL4 project it was discovered that the team did not directly, and solely, leverage `Isabelle`, via Isar proof script, to design the microkernel. Instead the team leveraged the functional language of Haskell for prototyping. The team showed how Haskell can be generated to Isar script, which allows verification by `Isabelle` [21]. As the team once discussed: "Given the precise semantics of the Haskell language, and the lack of side-effects of functional languages in general, it is a much simpler task to extract a formal model of the kernel compared to typical low-level systems languages like C" [21].

Subsequently, the Haskell implementation allowed for the automated generation of into Isar, documentation with LaTeX, and a compiled Kernel Prototype via the GHC (Haskell) compiler. The combination and relative ease of simultaneous development likely drove the usage of this process for the seL4 teams. Additionally, the ability to "code" and design in Haskell, with the code converting to Isar for `Isabelle` processing was likely a huge motivation due to the complexity of proof design and verification. The process allowed kernel properties to be achieved through functional programming implementation, and gave both the means to verify its model with `Isabelle` while additionally providing a prototype of the theory in a generated, binary form. Figure 2 shows the overall approach of the seL4 design process.

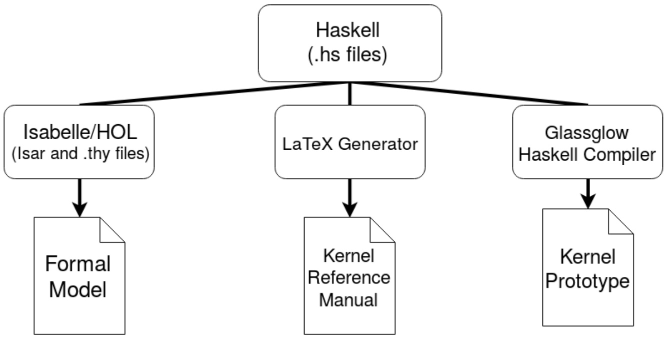


Figure 2. The Design Process for the seL4 Microkernel

The investigation into such tools and methods does not preclude the identification and application of other non-Isabelle tools. The intention of modeling a formal process after seL4 is to follow a known format or approach as to gain familiarity with the topics, which ultimately can help inform and decide on an appropriate approach for Android. Currently, no automated Haskell to Isar tools allow a Haskell implementation to be converted for analysis verification. As previously discussed, the seL4 team used Haskell to Isar theory files, and both are included in the GitHub.com hosted repository. It was not determined if the Haskell conversion scripts were included in the repository. Yet, if they were it is likely they are too specific, with respect to the seL4 source implementation, to use generically for other cases. (i.e., no clear documentation was found that showed the generation of Haskell to Isar theory files).

A seL4 Proving Environment with Docker

In May 2017 the seL4 follow-on team at Data61 released a new process to enable rapid deployment of a seL4S development environment. The process focused on a step-by-step tutorial on Linux that enables and executes Docker-based repositories. These repositories, once started, self deploy and setup the remainder of the tools and environment configurations for seL4 development. Additionally, this environment grabs the seL4 source files, and these are then used to execute and run the proof scripts and tools. These tests allow developers to directly access and examine the methods and sources for both projects, which then can be validated or extended to create new seL4-based applications.

Ultimately, the goal of this build-environment was to abstract the tool and isolate the programs from the host machine (i.e., OS updates and changes do not impact the development environment), and to allow the direct control and restriction of dependencies for the build environment [22]. A side-effect from the change in the seL4 development environment was the ability to rapidly and consistently deploy a development environment that was shared to the community. Instructions provided by Mondy can be easily followed to create a new Docker container with all needed development tools for seL4 [22].

The Use of Haskell Prototypes and Haskabelle in seL4

The seL4 team leveraged Haskell as a method to implement their theory into an executable format. Klein stated that Haskell was leveraged in part to translate a prototype of the kernel into `Isabelle` and to prove a large number of invariants and theorems [23]. In this method a specific and desired security feature or even prototype was implemented, converted, and subsequently proven in `Isabelle`. This implies that for such a method to be leveraged for A^2E^2, then each individual property needs to follow a similar methodology. This could be achieved, but a great need for familiarly with both Haskell and `Isabelle` are a requirement for the individual or team attempting to prove the A^2E^2 security properties. Evidence for such a tedious process was documented by Haftmann, one of the seL4 team engineers/developers, since it was stated that there is a constant need in developing ad-hoc conversion scripts that fit to a very specific setting [24]. However, this main approach of leveraging Haskell, converting into Isar, and proving the design with `Isabelle` was the main approach taken by the seL4 team.

However, Haftmann describes that `Isabelle` could generate Haskell, but it may be desired for Haskell to be generated into Isabelle-readable content instead, and one of methods in which to achieve this is with `Haskabelle` [24]. As the amalgamation of the name implies, the `Haskabelle` tool works by converting Haskell source code into Isar proof script (as stored in .thy files). As of 2010 the seL4 team had not been using `Haskabelle`, since they had constructed their own tool(s) to translate their Haskell implementation into an `Isabelle` readable format [23]. Around this time the seL4 team had seen promising experiments to indicate `Haskabelle` might be useful to `Isabelle` users, and had proposed two types of approaches: 1) program in Haskell, import into Isar theories for `Isabelle`, and prove the desired properties, and 2) a one-time import of an existing Haskell project into Isar, develop in Isar, and as needed produce Haskell using the built in `Isabelle` code generator [24]. In effect the seL4 team had identified or used two types of approaches: a direct Haskell to `Isabelle` procedure, and recommended Haskell to `Haskabelle` to `Isabelle` method. Therefore, to potentially leverage the proven security properties of the seL4 Kernel, while following an accepted security design with formal methods, the usage of `Haskabelle` in conjunction with `Isabelle` is reviewed when approaching theory design and verification. Each approach was deemed as a having potential interest as to applicability to this research and shall be expanded upon in Chapter 3.

II.2 Software Containment

Isolation and containment of software is not a new idea, and has been proposed in a variety of mediums over the past sixty years [25][26][27][28]. Examples of containment also include: the tool change root or chroot, virtual machines with type 1 (runs in an operation system) and type 2 (runs directly on hardware) hypervisors, and Linux Containers (LXC) including Docker [29]. Typically an operating system acts as the host, and software applications run identically (as they would on the host operating system) within a container.

Another example of containment is with a software container, since it is a complete runtime environment that supports runtime execution (i.e., the applications, dependencies, and other required binaries and configuration files are collected in a single execution space) [30]. A container has no knowledge of any processes or applications out side of its runtime, and everything the application(s) within the container need are also housed inside the container [31]. Containers that allow a direct virtualization of applications are referred to as operating system virtualization, and these types of isolation software are commonly referred to as containers [29]. Containers typically do not fully virtualize a full operating system, and will be a clone or instantiation of a portion of the host machine; this is one factor that separates a container from a virtual machine, since virtual machines are fully contained operating systems that are controlled with a hypervisor.

Many containment solutions have software that runs on top of the host operating system with varying levels of isolation. This isolation could be accomplished for numerous reasons, but commonly include allowing scalability of applications (e.g., cloud computing), protecting both running applications from one another and keeping the operating system away from harmful applications (e.g., crashes, exceptions, and even malicious applications). Advantages of ensuring safe and reliable execution has been documented by Amazon Web Services engineers the importance of formal specification and model checking [32]. Yet, it appears that there is no major or commercial product for containment that has been developed with formal methods.

Containers vs. Sandboxes

There are varying levels of isolation of software from its host, and a related form of software isolation is an application sandbox. The isolation techniques for many containerization solutions are based on: access controls, logical separation of the root directory with a chroot-like mechanism, Docker, and virtualization (e.g., VirtualBox, Amazon Web Services) [33].

Sandboxes, like containers, attempt to control the flow and impact of its controlled software on a host machine; for instance a sandbox would be used to control the safe crashing of its software or reduce the ability of its software from deadlocking the host machine like a container. Yet, sandboxes differ from containers, since a container seeks to have all needed runtime files in the same location of the isolated software, and a sandbox usually allows access to files directly on the host machine. Also, the software within the sandbox could still have unrestricted access to the rest of the system similar to a standard, host machine program (i.e., sandboxed applications can still access other locations and data on the host machine). Commonly, sandboxed software does not include or copy all files required files for its run time in an isolated space. In effect, a container is a more restrictive and isolating sandbox, since containers want to prevent most dependencies from and access to the host machine.

Current Containment Solutions

Various approaches have been presented to solve the problem of containing and securing software execution [34][35][36][37][38][39]. Over this period of research many solutions have been found, so a list of current containment solutions was collected relating to operating system level containment. Table 3 details containers and features of their containment.

Table 3. A Selected List of Containment Technologies

Application Name	Features
ARMlock	ARM processors use of memory domain support to create sandboxes
ARMor	Dynamically translation of code to execute on hardware implementing different memory modes
Boxify	Non-modifying Android App Encapsulation
Capsicum	A lightweight OS capability and sandbox framework for FreeBSD
Docker	OS-level application containment and virtualization on Windows/Linux/OSX
Inktag	Hypervisor; access controls, system crash recoverability
Linux-VServer	Virtual private server that securely partition resources on a Linux
LXC	Operating System-level virtualization that enabled multiple, concurrent Linux systems
MBOX	A sandbox mechanism to control host file system access and system call invocations
MiniBox	An x86 sandbox that protects the OS and sandboxed application
OpenVZ	A Linux-only virtualization system running a specific kernel
PREC	Android system call isolation to protect root exploitation
TxBox	Parallelized, speculative execution of untrusted applications
Vx32	Application-level virtual machine to run applications at a user-mode level

Current Containment Solutions Lack Assurance

Modern programs and applications have been created with good design processes and practices, which enforce desired boundaries of execution. Based on review of current literature no current containerization solutions have formally proven security properties, so there is no formal assurance of their specification or designs. At this time none of the solutions in Table 3 use of formal methods for either the design or their implementation. The lack of formal methods implies there is no proof that all discrete,possible states (or even the majority of states) of desired code execution are enforced or bounded (i.e., only specific unit and regression tests have been applied as opposed to general classes of solution). Additionally, the lack of formal methods implies the true assurance of security for the application cannot be determined, since the program is not mathematically proven and validated for a specification. This is an important fact to acknowledge, because if claims are made for the security of software they also must be paired with heavy caveats to the conditions and environment in which the program executes. Interestingly, this common lack of assurance was pointed out by researchers twenty years ago, because they were seeking to understand how software became as reliable as it is without having formally proven designs [40].

Using Docker as a Model System and Approach

Docker is container product that is mainly used on Desktop and Server personal computers, and is created by Docker Incorporated. It is open source software hosted on GitHub, which allows for the tool to be analyzed, reviewed, and even modified. There is no direct port of Docker that runs on Android, and this is likely due to the fact that most applied uses of Docker are command-line-based applications. At this time the software product is not formally assured, so there are not direct assurance capabilities for the present research to leverage. However, this research considers how Docker provides isolation as a model to support the requirements definition of an assured container, since the containment and isolation that Docker leverages are potentially useful on Android.

A close comparison of how Docker implements a containers is the tool Chroot. Docker and Chroot are implemented in different programming languages, but both appear to leverage the Linux Kernel to implement permissions. Docker appears to provide many features to support isolation, but the company also states that these features are at a high-level and specifics are not documented [41].

When looking into some of Docker isolation techniques it was discovered that the tool uses a combination of LXC (Linux Container) or runC (a command line tool based on the OpenContainer specifications), and the Advanced Multi-layered Unification Filesystem (AuFS) tools to achieve its containerization goals [42]. The specifics to how each of the tools are leveraged is not covered in this research; a simpler goal of general isolation methods is being sought as opposed to implementation specific designs for containers. Docker features many low-level activities specific to the OS (e.g., permission enforcement with the Linux Kernel), since many of the higher-level capabilities focus on user interaction with Docker and configuration management of Docker containers (e.g., container provisioning and records management). It can be argued that the primary tool to enable low-level activities in Docker was LXC (i.e., Docker was eventually migrated from LXC to runC due to architectural and open standard issues versus capability concerns [42]). LXC technologies and features were found to be the utilized as an interface and collection of applications for containment with the Linux Kernel. The use of LXC and the Linux Kernel was described by Moser, which included Linux features of: kernel namespaces, Apparmor & SELinux profiles, kernel functionality (e.g, sys_boot, sys_chroot, sys_module, net_bind_service, and sys_mknod), control groups / cgroups, Chroot jails, and seccomp policies [43]. These features are of the utmost interest, since they appear to parallel the focus of Android security concerns.

The Go Programming Language

Docker is implemented in the Go programming language, and there are development tools that allow the creation of Go programs for Android. The Go Programming Language (Golang) is considered a new programming language since it was created in 2007 with an open source version being released in 2009 [44]. Version 1.0.0 was released in 2012 and many new versions and updates are being consistently released [45]. Yet, the primary focus of this programming language is for application development on desktop and server environments.

Thus, if a Go container would need root level accesses, then it is probable the container would prevented from executing on Android due to the standard security model (i.e., all non-Google and non-OEM applications are prevented from root-level execution). Conceivably, if a container application were to be developed in Go, that works on Linux, then access levels could be permitted for the container (e.g., if an OEM were to include it with its packaging, or if a "rooted" device were to execute the software). Specific runtime issues were not examined in this research, since a focus is how to assure the specification of security properties and their implementation into programs as a whole.

An Assured, Docker-like Linux Container Implemented in Golang

There are several examples on-line of creating Linux kernel enforced (i.e., namespace) containers, and examples can be found that implement a container in a minimum amount of code and even as in as little as 500 lines of C code [46]. Developers are claiming to have achieved Linux containers in 100 lines of Go code (i.e., Go as a higher level language should take less C code to implement). One such example of a Go container is by Friedman, which was covered by Rice during a 2016 conference for Container Camp UK [47]. Both versions of the source code can be found on-line, and show two slightly different approaches in the implementation of Go for a Linux, chroot-like container [48][49]. These source files show different settings and configurations (e.g., base directories, permissions, etc.), but at the core of both programs is the use of Linux Kernel features for containment. The simple design is beneficial since it may reduce any potential issues if such code could be ported to Android.

Additionally, having these example Go implementations of containers as a starting point may simplify assurance of implementation verification. As previously discussed, KLEE is a symbolic execution tool, but it focuses on C and C++. Thus, an open research question forms with the ability of a Go language container to invocate C and C++ functions, which may allow KLEE to be used for symbolic execution in a Go container. The successful implementation of KLEE into the development of a Go container would allow for formal methods to be applied to the implementation of this software. More details on this approach will be discussed in Chapter 3.

Migrating Go Linux Applications to Android

In July 2014 the first usable "Go Mobile" example (based on a C/Java Native Interface and logcat) was committed into the main mobile repository for Go. This suite of tools allows for cross compilation of Go source code to both iOS and Android devices. The Android versions of the code for Go is compiled to native (usually Arm-base) machine binaries, which can be executed by the Linux Kernel in Android.

"Go Mobile" offers a means to allow Go development that can easily be ported to Android, thus if a container was designed with Go it may have the capability to be ported to the mobile operating system. At this time, however, it is unknown the level of features and accesses on Android that Go code may execute. On most Android implementations there are restrictions for the root user and root-level API function calls. Go has no root-based Linux kernel function restrictions, since an application on workstations and servers application would authenticate an application prior to execution.

The Go Mobile suite of tools allow for Go development in the Android Studio IDE. With respect to a Go-based coded application for Linux the tool suite "Mobile" from the experimental go tool chains can be leveraged to allow for cross compilation of go source code for Android. Due to the complexity of this research only the base examples included in the Go Mobile code based were tried. The tools allow for code to be complied into into an Android Package Kit (APK), which can be transferred and installed on Android. These applications execute similarly to Native Development Kit that is used to build C-language binaries. Meaning that a user's Go code can be built and innovated directly, with message or data passing, as ARM binary file (i.e., Go code is built and can be called like a method from a Java-based application), and the Go code could also be invocated and started without having access to the primary Android OS features (e.g., intents).

It should be noted that the Go Mobile tools require the NDK to be installed for Android. This can be done directly from the archived package binaries, or by the method of the Android SDK Manager. For ease of installation the NDK was installed with Android Studio, and a path modification for BASH was added for both the binary folders of Android Studio and the sub-folder for the NDK. Once the development kits are properly installed and configured the installation of the Go Mobile tools can be accomplished with the commands of: *go getgolang.org/x/mobile/cmd/gomobile* and then *gomobile init*. At this point a command line environment is established and configured for Android APK generation. The Go Mobile project includes two main examples that display simple OpenGL and bind() elements for Android, which can be modified for use in Go programming implementations. While the core Go library is functional on Linux there are many instances of API/function calls requiring root for proper execution; this is not usually permissible on Android due to the locked-down nature and permission model. Hence, at this time it is unknown the supportability of the extended Go library on Android. Additionally, there were attempts at launching the Go Mobile examples on an unlocked and installed Cyanogenmod ROM of Android, which grants full root-level permissions of an Android device. However, the Go-based system calls, the Android monkey tool, and the development tool "am" could not successfully launch function calls at a root level. These attempts included an integrated version of Rice's Go container code targeting an SD Card storage device, but all system calls failed. It is presumed that this was due to the Go containers needing root permissions, but it the attempts were also unable to print or log any errors relating to the failed attempts. These attempts are additionally presumed to fail in due part to the unfamiliarity of leveraging Android permissions along with the unknown nature of Go code executing within the Android environment. However, it should be noted that simple Go applications would execute as expected (e.g., computational or database type input/output operations), so to have issues resulting from low-level permission required activities is not surprising. Thus, if a new means of userspace containment could be found, or enabled in the Linux kernel, then it is presumed that a Go container like Rice and Friedman could be implemented.

II.3 Android

Mobile devices (e.g., cellular phones and tablets) are typically not considered to be a personal computer (i.e., a desktop or laptop) by an average consumer. This is likely due to the devices having a considerable amount of differing hardware than a personal computer to miniaturize the devices to allow for portable computing and cellular phone service. However, a common factor between personal computers and mobile devices is both types of computers run an operating system. An operating system is software that controls and directs the operation of a computer and its programs [50]. New operating systems have developed over the past few decades to enable mobile devices as portable computers.

The increasing capability of hardware and batteries gave rise, in 2003, to Android, Incorporated [51], which sought to develop smarter mobile devices to be more aware of the owner's location and preferences [51]. Then in 2005 Google, Inc. bought Android, Inc., created The Open Handset Alliance in 2007 (focusing on open standards for mobile phones), and released the first Android Software Development Kit (SDK) in 2007 as well. Release of the official Android operating system, simply 1.0 [52], occurred in October 2008 with the near simultaneous release of the operating system's source code [53] and the HTC Dream (G1) phone [51]. A continuing theme of Google's acquisition and release of Android has been open source code and design, which is pivotal in the analysis and development of software for devices running the operating system. In the ten years after the release of the Android SDK the operating system now has over 2 Billion monthly active devices [2], and in 2017 trends show that Android, based of web traffic trackers, has taken over the lead from Windows as the most popular operating system [2].

The Android Security Model

An interesting fact of Android is the central design around the Linux Kernel. Kernels can be thought of as the brain for an operating system, since kernels direct the flow and control of internal and external devices and programs. Many security features on Android relate, or are enabled due, to the Linux kernel. The kernel acts as a base upon which Android developers have expanded upon to ensure operating system and subsequently application security. A model of the Android OS can be seen in Figure 3 showcases the documentation, and most of security is focused at the lowest two layers.

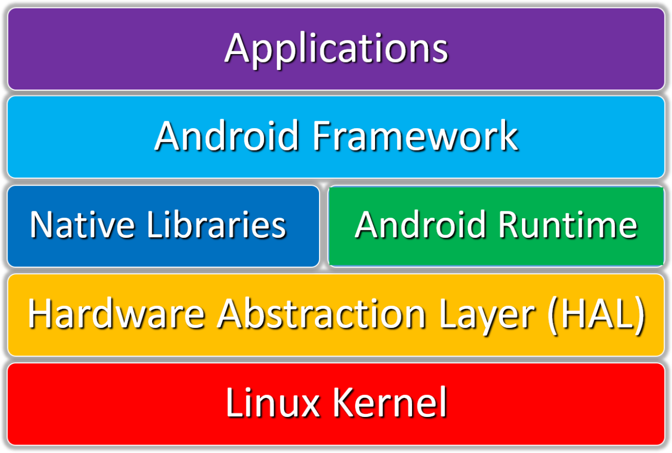


Figure 3. The Android Software Stack

This software stack model has not changed much over the duration of Android's development, but the implemented security features have been ever evolving over the past ten years. As new Linux Kernel features are added they are potentially integrated into Android. Features beyond the Linux kernel appear to come from the community and core developers of Android that are expanding best practices of security features. A summary of security features, per major Android Release, can be found in Appendix B.1.

Android Sandboxing

At the time of this writing there are major shifts planned for Android 8.0 and beyond. However, the major trend across all of the previously added or planned improved security features is a continued focus of Kernel-based protections in conjunction with sandbox containment of application [54]. Android's main security goal appears to be to protect the central capabilities of Android's Linux kernel [55], while ensuring the applications cannot forcibly interact with other applications nor the hardware to gain access to the Linux kernel [54]. The configurations and protections in place with the kernel allow a focus on making applications run within sandboxed portions of Android, which prevent the rogue application from harming other applications, the Android system, or the device itself [55]. Android's approach to sandboxing, of both data and code execution, is known as the Android Application Sandbox [56]. A unique feature of Android is the fact it assigns a unique User IDentification (UID) number to each Android application and runs it as that user in a separate process, which in turn enables the sandbox to enforce security between applications and the system at the process level [55]. Again, via Linux kernel features (e.g., user and group IDs) applications are enforced to a security model extends to native code and to operating system applications [55]. An interesting note regarding the sandbox technologies is they provide applications with an expectation of isolation from other processes on the system which includes root processes and debuggers [57]. Furthermore additional Android best practices include: non-access to data within individual application data folders (unless via debugging), non-access to memory of applications (unless via debugging), and devices must not include any application that accesses data or memory of other applications or processes.

II.4 Existing Security Models and Frameworks

A selection of existing security models and frameworks are leveraged in this research to identify a common set of security properties needed for assurable specifications. Specifically, the models of the Confidentiality, Integrity, and Availability (CIA) Triad, The Open Group's Open Information Security Management Maturity Model (O-ISM3), the Department of Defense (DoD) Security Requirements as derived from the National Institute of Standards and Technology Special Publication (NIST SP) 800-53, and an analysis of the Android Application Sandbox's security features are leveraged for this research. The combination of these four models have been selected specifically due to: the general information security practices, commercial usage, United States security direction, and use on Android. There are many models for security, but the four selected models appear to have appropriate coverage needed for general and specific properties needed for a comprehensive security design.

The Confidentiality, Integrity, and Availability Triad

A cornerstone in security design has been the CIA Triad, which is comprised of the cornerstone information security topics of Confidentiality, Integrity, and Availability. These three categories are used to describe general requirements of security design, and also show the interaction of the synergistic and competing areas. For example if a system was designed to be confidential then there likely are tradeoffs for availability to a user; such a tradeoff may have a user needing to login and verify access to a system, but if the user was unable to verify with the system then the availability of system access is reduced. These three types of categories are used as a general rule, since there are no specific metrics or measurable means to verify the impact of a system's design choices. Many first order CIA solutions act more as labels for security categories as opposed to implementable security features, but these solutions still drive first order design choices for a system's security design. Also, it should be noted that the category of availability by itself offers no explicit security features, since availability is normally a measure or a capability of system up time or accessibility. Therefore the two main topics of the triad under review are confidentiality and integrity.

There are many classes of security features that could be categorized or applied to the CIA triad. Security features of interest can derived from the the two main triad topics; for instance, the security features of data protection (confidentiality), data transmission (confidentiality) and data assurance (integrity) are usually important to a secure design. Cryptographic means are leveraged to implement these three security features (e.g., cryptographic one-way hashes are used to validate data integrity, and encryption is used to ensure data protection at rest and when transmitted). Yet, it should be noted that cryptographic solutions are not the only means to implement confidentiality and integrity. Access controls, for instance, are very common in the separation of user spaces and data (e.g., separated user name spaces, file system partitioning, and user based permission systems), and these could be used to implement confidentiality and integrity.

Furthermore, the three existing categories can be extended from the triad to add more security categories. Two security properties that can be extended into the CIA Triad are Authenticity and Non-repudiation. Respectively, these two properties typically focus on adding 1) communications and access verifiability, and 2) system logging and traceability capabilities. Similar to the previously discussed topics, of confidentiality and integrity, authenticity and non-repudiation are commonly implemented by cryptological means (e.g., users can be validated with passwords and shared secrets, and when system resources are accessed a data record could be kept to ensure the proper access is being maintained). These extended categories offer additional approaches to ensuring a system's security properties, but again are more applicable as labeling mechanisms versus true security properties that are implementable.

The high-level abstraction of CIA Triad lacks explicit security features, and focuses more on the interplay of the categories. Thus, the CIA Triad lends itself to derive security properties in prototyping or generating specific security features. However, to generate security requirements and properties for a design more specific models are needed for concrete examples and definitions of security properties and features.

The Open Group's Open Information Security Management Maturity Model

The Open Group sets information technology standards across 500 member organizations in order to help standardize emerging requirements, establish policies, share best practices while integrating and facilitating open source technologies [58]. As such The Open Group has created a technology neutral, and business requirement driven, set of standards to help address security for information technology. This standard is the O-ISM3 and it aims to ensure implemented security processes are consistent with an organization's business requirements through the identification of relevant security controls and processes.

O-ISM3 encourages the formal measurement of effectiveness of each security management process with the identification of four separate process levels. These levels are 1) Strategic (broad goals, coordination, and provision of resources), 2) Tactical (design and implementation of the "Information Security Management Systems" specific goals, and management of resources), 3) Operational (the means to achieve via technical processes), and 4) Generic (for general management relating to tracking, implementation, status, etc.); note: these process levels appear to echo military definitions, but they are not, in fact, the same. The first three process levels seek to take a large security concept and hone it down to a specific method that is implementable for an organization. For example the issue of "privacy" at a Strategic level, could be separated into data privacy for customers and also data privacy for employees at a Tactical level. Subsequently, the different approaches to privacy for users and employees would have different implementations at an Operational level (e.g., encrypted storage for users, and access controls with separate work spaces for the employees). A way to think about the O-ISM3 is not in terms of raw properties of security and the means in which to enable the features, but the way to think about the model is how to apply something like the CIA Triad to an organization through the method that is O-ISM3. The Open Group states that the objectives of the standard are to: provide an approach for creating Information Security Management Systems (ISMSs) (for the business's mission and compliance needs), provide an approach for any size organization, enable a way to optimize investments into information security, enable continuous improvement of ISMSs using metrics, and enable metric-driven, verifiable outsourcing of security processes.

As such, the O-ISM3 has real contributions to information security and security property design, since it emphasizes that any implemented feature can be measured. Specifically, the model states that a metric is a measurement that can be interpreted and investigated by comparing it with a series of previous or equipment measurements, and by improving such a metric the total process has value added [58]. Furthermore, the model states that when the metrics are used to improve the consistency of the process, and if variations occur in the metric, and are identified, then the process metric improves as well. Such metrics include activity (the ISMS's inputs and outputs), scope (how many of the input types are being used), effectiveness/availability (comparison of fraction of inputs that produce an output), quality (comparison of the ISMS to an ideal output), load (the budgeted vs. actual consumption of resources), and efficiency (the load over time) [58]. The specifics of the metrics are left to the implementer, but it is recommended that SANS, NIST SP 800-55, and ISO/IEC 27004:2009 should be used to design metrics to support security governance [58].

The main contribution of O-ISM3 is the highly detailed list of 45 Processes that should be considered in the design and implementation of an ISMS. The breakout for the respective process categories are: 3 Generic Processes, 4 Strategic Management Processes, 12 Tactical Management Processes, and 26 Operational Management Processes. Table 4 is the list of all main, available O-ISM3 Processes.

Table 4. O-ISM3 Processes

Process Type	Process Subcategory	Process Name
Generic		Knowledge Management
Generic		ISMS and Business Audit
Generic		ISM Design and Evolution
Strategic		Report to Stakeholders
Strategic		Coordination
Strategic		Define Division of Duties Rules
Strategic		Allocate Resources for Information Security
Tactical		Report to Strategic Management
Tactical		Manage Allocated Resources
Tactical		Define Security Targets & Security Objectives
Tactical		Service Level Management
Tactical		Security Architecture
Tactical		Insurance Management
Tactical	Personnel Security	Background Checks
Tactical	Personnel Security	Personnel Security existing employees
Tactical	Personnel Security	Security Personnel Training
Tactical	Personnel Security	Disciplinary Process
Tactical	Personnel Security	Security Awareness
Tactical		Information Operations
Operational		Report to Tactical Management
Operational		Security Procurement
Operational	Life Cycle Control	Inventory Management
Operational	Life Cycle Control	IT Managed Domain Change Control
Operational	Life Cycle Control	IT Managed Domain Patching
Operational	Life Cycle Control	IT Managed Domain Clearing
Operational	Life Cycle Control	IT Managed Domain Hardening
Operational	Life Cycle Control	Software Development Life Cycle Control
Operational	Life Cycle Control	Security Measures Change Control
Operational	Life Cycle Control	Segmentation and Filtering Management
Operational	Life Cycle Control	Malware Protection Management
Operational	Access and Environmental Control	Access Control
Operational	Access and Environmental Control	User Registration
Operational	Access and Environmental Control	Physical Environment Protection Management
Operational	Access and Environmental Control	Physical Environment Protection Program
Operational	Availability Control	Back-up Management
Operational	Availability Control	Operations Continuity Management
Operational	Availability Control	Enhanced Reliability and Availability Management
Operational	Availability Control	Archiving Management
Operational	Testing and Auditing	Internal Technical Audit
Operational	Testing and Auditing	Incident Emulation
Operational	Testing and Auditing	Information Quality and Compliance Assessment
Operational	Monitoring	Alerts Monitoring
Operational	Monitoring	Internal Events Detection and Analysis
Operational	Monitoring	External Events Detection and Analysis
Operational	Incident Handling	Handling of Incidents and Near-incidents
Operational	Incident Handling	Forensics

The O-ISM3 states that of 16 fundamental processes should be considered essential to any initial in a top-down implementation [58], which means only some 35% of the total primary processes are needed for a minimum security perspective. These processes can be found in Table 5.

Table 5. Down-selected, Essential O-ISM3 Processes

Process Type	Process Subcategory	Process Name
Generic		Knowledge Management
Generic		ISM Design and Evolution
Strategic		Report to Stakeholders
Strategic		Coordination
Strategic		Allocate Resources for Information Security
Tactical		Report to Strategic Management
Tactical		Manage Allocated Resources
Tactical		Define Security Targets & Security Objectives
Tactical		Service Level Management
Operational		Report to Tactical Management
Operational	Life Cycle Control	IT Managed Domain Patching
Operational	Life Cycle Control	Segmentation and Filtering Management
Operational	Life Cycle Control	Malware Protection Management
Operational	Access and Environmental Control	Access Control
Operational	Availability Control	Backup Management
Operational	Testing and Auditing	Information Quality and Compliance Assessment

Department of Defense Security Requirements as Derived from the NIST SP 800-53

A different approach to security, from commercial enterprises and entities, is being led by the United States' Department of Defense with respect for security system design and implementation. Classically, the Department of Defense places a great emphasis on Systems Engineering, which has also had a major focus on protection and security of the system being designed. Analysis of Department of Defense security requirements is being considered due to the specific environment of security products from the outset. Additionally, a majority of the documentation and security design is openly available due to the extremely documented approach of their acquisitions processes for contracting purposes. Such documentation provides specific, contractually motivated, design and development considerations are leveraged in this body of research.

Before looking at the specific DoD security requirements, a quick review of the catalysts and origins of Department of Defense processes are discussed. The first rule setting body for acquisitions in the United States starts with the Code of Federal Regulations (CFR), which is an annual codification of the general and permanent rules in the Federal Register from the executive departments and its respective agencies [59]. Within this code there are 50 titles containing a broad range of activities relating to regulatable activities, and specifically Title 48 is the Federal Acquisition Regulations System [60]. The basis for all United States Federal acquisitions and contracting begins with the Federal Acquisition Regulations Systems (FARS) being Chapter 1 in Title 48. Subsequently modeled after the FARS is Chapter 2 being the Defense Federal Acquisition Regulation Supplement (DFARS), which are regulations specific to military acquisitions for the Department of Defense. Additionally, Chapter 2 is comprised of nine sub-chapters (having parts 200-299) and nine appendices, and a specific sub-chapter of interest to system security properties is subpart 239.71 entitled "Security and Privacy for Computer Systems"; the section states it focuses on information assurance and Privacy Act considerations. Furthermore, the section states that information security are the measures that protect and defend data entered, processed, transmitted, stored, retrieved, displayed, or destroyed [61]. According to this portion of the DFARS, assurance occurs on information systems through ensuring availability, integrity, authentication, confidentiality, and non-repudiation via protection, detection, and reaction capabilities [61]. Additionally, compliance with the approach is directed to the sub-organizations of the Department of Defense through the documents in Table 6, and it should be noted that in 2013 and 2014 a shift of terminology occurred in the DoD that moved from Information Assurance to Cybersecurity.

Table 6. DFARS: Required Implementation Documents for Information Assurance

Document Title
The National Security Act
The Clinger-Cohen Act
National Security Telecommunications and Information Systems Security Policy No. 11
Federal Information Processing Standards
DoD Directive 8500.1, Information Assurance
DoD Instruction 8500.2, Information Assurance Implementation
DoD Directive 8140.01, Cyberspace Workforce Management
DoD Manual 8570.01-M

At first glance the sum total of the regulation aligns the Department of Defense to leverage the CIA Triad as the base security model for all systems. However, over the last several years many of the directive documents have be updated or modified from the focus on "Information Assurance" to that of "Cybersecurity." This shift is explicitly covered in the Department of Defense Instruction 8500.01, which its purpose is to provide guidance on how to protect and defend Department of Defense information and information technology. Yet, the most important detail is that a majority of the 8500.01 is modeled from National Institute of Standards and Technology Special Publication 800-39 and Committee on National Security Systems Policy 22.

Upon review of both the NIST SP 800-39 and the CNSS Policy 22 it was discovered that there are yet more driving documents when defining security properties for a system. The NIST SP 800-39 has listed the publications of 800-37, 800-53, 800-53A, and 800-30 are the definite series of security standards and guidelines necessary for managing information security risk [62]. Within this final set of NIST regulating documents it was discovered that Special Publication 800-53, titled "Recommended Security Controls for Federal Information Systems and Organizations," has included concrete security properties and implementable means for security. Also, the CNSS Policy 22 led to the discovery of a synergistic document to the 800-53, which is the CNSS Report 1253 entitled: "Security Categorization and Control Selection for

National Security Systems and it states it is modeled after the NIST 800-53. The intensive trail of regulation investigation resulted in two concrete security property sources for the Department of Defense: the Security Control Baselines in the CNSS Report 1253 and the Appendices D through F in the NIST SP 800-53. However, only the NIST SP 800-53 is considered in evaluating security properties, since the CNSS report states that it is an extension/implementation based upon the NIST SP 800-53. Yet, as a side note: if a researcher was curious as to see the mapping or overlay of NIST security Controls to CIA the Appendix D in CNSS Report 1253 includes tables indicating this information. Ultimately, The Department of Defense is trying to seek adequate security for its systems, and security is defined as: protective measures that are commensurate with the consequences and probability of loss, misuse, or unauthorized access to, or modification of information [63]. The NIST SP 800-53 protective measures have been placed into 18 major categories with a total of 256 controls, which varying across the main categories [64]. A summary of the categories and their tallied amounts of controls can be seen in Table 7.

As seen with the O-ISM3, not all of the above categories are implementable into a direct system, since some of the controls are procedural or information-based only (e.g, Awareness and Training deal with persons using a system, and Program Management and Risk Assessment deal with the administration of a system). Also, not all systems need every single security control, and in-fact the minimum/low-impact baseline recommended by NIST only covers about 45% of the 256 total controls (see Table 8).

The specifics on how to categorize a system are not included in this research, but if desired the information can be found in the FIPS Publication 199. Hence, a generalization of how to categorize a system could be from extending and applying the definitions of FIPS 199 system loss: 1) a system is low if loss of CIA could

Table 7. NIST SP 800-53 Control Categories and Tallied Controls

ID Code	Category	Amount of Controls
AC	Access Control	25
AU	Audit and Accountability	16
AT	Awareness and Training	5
CM	Configuration Management	11
CP	Contingency Planning	13
IA	Identification and Authentication	11
IR	Incident Response	10
MA	Maintenance	6
MP	Media Protection	8
PS	Personnel Security	8
PE	Physical and Environmental Protection	20
PL	Planning	9
PM	Program Management	16
RA	Risk Assessment	6
CA	Security Assessment and Authorization	9
SC	System and Communications Protection	44
SI	System and Information Integrity	17
SA	System and Services Acquisition	22

Table 8. NIST SP 800-53 Minimum Recommended Security Controls

Desired Security Control Level	Recommended Controls	Coverage Percent
High-Impact Baseline	170	66.41%
Moderate-Impact Baseline	159	62.11%
Low-Impact Baseline	115	44.92%

be expected to have a limited adverse effect on the organization (the organization is able to perform its primary functions, but the effectiveness of the functions is noticeably reduced), 2) a system is moderate if loss of CIA could be expected to have a serious adverse effect on the organization (a significant degradation in mission capability to an extent and duration that the organization is able to perform its primary functions, but the effectiveness of the functions is significantly reduced), or 3) system is high if loss of CIA could be expected to have a severe or catastrophic adverse effect on the organization (a severe degradation in or loss of mission capability to an extent and duration that the organization is not able to perform one or more of its primary functions) [65]. Therefore, with the application of applying minimum security properties to a container environment, with the absence of administrative type-controls, can be seen as the of controls in Table 9.

Android Application Sandbox's Security Features

Finalizing the review of security system properties is a quick overview of Android Application Sandbox features. This sandbox is applicable to the implementation of Assured Android Execution Environments, since it appears to provide many desirable features for a container. Table 10 shows the security properties and features the Android sandbox currently supports.

Unfortunately, the primary documentation for Android developers, on the Android source web page, only lists the security features in the previous table. There was no discovered, detailed analysis for the motivation and designs of the features, nor were there any formal analysis found on the topic. The specifics for these properties per major Android release can be seen in Appendix B.1, but the major finding was a majority of the security features of Android are enabled or enforced by the Linux Kernel. This information is still useful even with the absence of implementation specifics, because they still provide the general outline and feature set for Android security. Thus, with these features rounding out the information the investigation into potential security properties a final consolidation and mapping of these properties can occur to help define general security properties.

II.5 Related Research

This section focuses on select research that relates to the overall design and motivation of this thesis. This previous work helped set the stage of understanding and future direction needed for provable security on Android. It should be noted that not all work is directly referenced or used, but the inclusion of the following research may be helpful for future researchers with respect to this topic.

Formal Modeling and Reasoning about the Android Security Framework

In 2012 a paper by Armando, et al., was released that reviewed and recommended a formal model for the Android security framework [66]. The researchers detail the Android Security Framework, present a formal representation of the framework, and present a type and effect system [66]. From research relating to formal descriptions of the Android's security the paper shows the most detailed modeling and reasoning for the operating system. This high-level approach strives to describe the features of Android formally at its different layers across applications and system features [66]. The proposed model was stated to guarantee any possible behavior a platform was to have at runtime with history expressions (the security-relevant side effects produced by computations conforming to explicit permissions) [66]. Ultimately, the results were proposed to be statically analyzable from the model, and the researchers had planned to verify the security properties in future work [66]. However, as of this writing the researchers have not released any follow-on work or applications to the Android code base.

Boxify: Stock Android Application Sandboxing

An extremely relevant paper was published in 2015 by Backes, et al., and it discusses the means to enable application virtualization and process-based privilege separation on Android to securely encapsulate untrusted apps in an isolated environment [67]. The researchers have shown they are avoiding a then common practice of application modification, which was used in tools such as Aurasium, I-ARM-Droid, RetroSkeleton, AppGuard, and DroidForce [67]. The two major motivations for their approach are the fact that to enable such containment on Android usually involves extensive modifications to Android and its application framework, and even if the tasks are accomplished the proposed solutions are rarely adopted by Google or the device vendors [67]. Thus, to avoid custom Android ROMS (e.g., formerly CyanogenMod and now Lineage OS) the developers have decided to implement their application directly on top of the Android Application Framework. The source code is said to be available by request for academic purposes, and there is a mention of the source becoming open once a licensing issue was resolved. However, the main page for Boxify was scraped by Archive.org's Wayback Machine in December of 2016, and as of a year later the website still has a forthcoming release of the source code. While seemingly an ideal candidate and fit for this research there is no concrete support of an open source license, therefore further research was not completed.

Java-based Implementation Assurance Tools

Going over a single tool, in a single application, is obviously non-inclusive of all appropriate methods for implementation verification. So far a single tool, and process, has been showcased for this research, but there were a couple of other tools that were discovered over this period of research. One of the biggest areas that were not discussed were the potential usage of Java-based tools and methods. Additionally, there were other discovered abstract interpreters that were found specific to Android.

Java Abstract Interpreters

It again should be reiterated that Android-based Java, it its resulting bytecode, are not the same as Oracle Corporation's Java and bytecode. This implies that the application and usage of the same source code results in different outcomes and likely different means of execution. Nonetheless, giving Java abstract interpreters a mention lends at least an identification of similar efforts ongoing in the world of implementation assurance.

Some of the first discovered efforts for assurance in Java execution were seen in the early 2000s with tools such as Java-MaC [68]. Sadly, over this period there are few cases of continuous development, and the tools seem to ebb and flow for popularity and usage. Even with this sporadic development there are at least two modern approaches to formal assurance on Java with the tools of `KeY` and `Soot`.

`KeY`, or the `KeY System`, is a formal software tool that has was started in the late 1990s used for the design, implementation, and formal verification for Java [69]. A main feature of interest is `KeY`'s Symbolic Execution Debugger, which is an integrated Eclipse IDE plug-in. This debugger constructs a proof and extracts the symbolic execution tree from the source code in a fully automatic way [70]. This allows a Java developer to leverage and utilize a form of abstract interpretation to assure their design meets the original specification, and if `KeY` is provided with proof specifications it can incorporate and validate such logic [70]. At this time there is no indication that `KeY` supports Android-based java, but if core features of A^2E^2 were designed in non-specific Java code (i.e., only methods and objects that are usable in both Android and standard Java) then it may be possible to leverage the tool as-is.

Another tool, `Soot`, may be the most promising candidate for the use of Java-based implementation analysis. `Soot` was created as a Java optimization framework, but overtime the tool has extended to support the analysis, instrumentation, optimization, and visualization of Java (source and bytecode) and Android (bytecode) applications [71]. The tool has been used for examining permissions, analyzing Dalvik bytecode, examining call graphs, and even symbolic execution for Android applications [72][73][74][75]. This tool seems to be more promising than `KeY`, since there is an active community with respect to Android. Time will tell to determine if such formal methods as abstract interpretation will continue with `Soot`, but hopefully this information will help future researchers in this area.

Current Android Interpreters

While there have been attempts at porting Java based capabilities to Android there also have been efforts to directly develop tools for the Operating System directly. In 2012 Symdroid is a product of the University of Maryland that was developed around 2012, and it was designed to be a symbolic executor for Android [76][77]. Unfortunately, not much beyond its development and usage for symbolic execution were discovered since there are no known source code repositories for the tool. However, it is included to show that such capabilities are in development and by extension of interest to the formal methods community.

A more promising Android set of tools in development are Android Extensions to the Java PathFinder (JPF) tool. `Java PathFinder` is a NASA tool that creates a new Java Virtual Machine in which java applications are executed to discover defects [78]. In recent years there have been efforts to add Android extensions into JPF, and the most notable extensions are `JPF-Android` and `PathDroid`. `JPF-Android` is a non-GUI based extension that looks at an application's logic with the detection of errors and unwanted behavior [79]. The tool appears to be in active development, but the specifics of how and what portion of the JPF it leverages were not determined. Also, the tool `PathDroid` is another extension of JPF, implements and emulates the Dalvik bytecode instruction set and virtual machine while re-using the JPF infrastructure [80]. This tool, based upon the last commits to the open repository of the code, appears to have stopped development in 2015.

A survey of Applied Formal Methods with Isabelle

An extremely interesting find was a survey completed by Blanchette and Popescu entitled "Isabelle and Security." The paper is very short at five pages in length, but had some forty-one references that detail security proof related work for `Isabelle` [81]. It was decided to include this work, since its extremely applicable for researchers if they are looking to potentially use `Isabelle` as a theorem prover. One of the first topics and work to be recommended is that of Nipkow and Popescu including a unified set of security concepts and type systems, which Blanchette and Popescu state as a simplified set of proofs of correctness [81]. It was discovered by Blanchette and Popescu that several cryptographic protocols were proven in `Isabelle` by Paulson and both shared and public key cryptography means were also proven (by Otway-REss and Needham-Schroeder respectively) [81]. An already discussed project that details integrity and data flow enforcement is the work by the seL4 team [81]. Additionally, the seL4 team contributed efforts toward access control proofs that can be found in their l4v verification efforts, and Brucker, et al., worked on the Unified Policy Framework toward access control [81]. With respect to concurrency assurance efforts Mantel looked at formal proofs for language-based security, and the efforts of Nipkow, et al., worked toward applications of multithreading [81]. Unfortunately, no explicit work was detailed for non-repudiated accounting and journaling, but work relating to noninterference may be applicable within the works detailed by Blanchette and Popescu. Blanchette and Popescu provide a source of network security with the work of Dickmann on network policy verification [81]; also, a non-survey item to be recommend is that of the Netfilter Iptables Firewall. Finally, with respect to data and execution flows, as to include information-flows, Mantel created a security driven input/output framework (i.e., the Modular Assembly Kit for Security) and the tool of Reliably Secure Software Systems that were formalized with `Isabelle`. Obviously, this select list of tools and features are not totally encompassing of the survey nor of that of the `Isabelle` body of work, but these programs may shed insights into the methods used with `Isabelle` to assure formal designs.

Table 9. DFARS: Minimum Security Controls for Safeguarding

Control	Type	Identifier
Account Management	Access Control	AC-2
Access Enforcement	Access Control	AC-3(4)
Information Flow Enforcement	Access Control	AC-4
Least Privilege	Access Control	AC-6
Unsuccessful Logon Attempts	Access Control	AC-7
Session Lock	Access Control	AC-11(1)
Remote Access	Access Control	AC-17(2)
Wireless Access	Access Control	AC-18(1)
Access Control for Mobile Devices	Access Control	AC-19
Use of External Information Systems	Access Control	AC-20(1)
Use of External Information Systems	Access Control	AC-20(2)
Publicly Accessible Content	Access Control	AC-22
Security Awareness Training	Awareness and Training	AT-2
Audit Events	Audit and Accountability	AU-2
Content of Audit Records	Audit and Accountability	AU-3
Audit Review, Analysis and Reporting	Audit and Accountability	AU-6(1)
Audit Reduction and Report Generation	Audit and Accountability	AU-7
Timestamps	Audit and Accountability	AU-8
Protection of Audit Information	Audit and Accountability	AU-9
Baseline Configuration	Configuration Management	CM-2
Configuration Settings	Configuration Management	CM-6
Least Functionality	Configuration Management	CM-7
Information System Component Inventory	Configuration Management	CM-8
Information System Backup	Contingency Planning Acquisition	CP-9
Identification and Authentication (Organizational Users)	Identification and Authentication	IA-2
Identifier Management	Identification and Authentication	IA-4
Authenticator Management	Identification and Authentication	IA-5(1)
Incident Response Training	Incident Response Integrity	IR-2
Incident Handling	Incident Response Integrity	IR-4
Incident Monitoring	Incident Response Integrity	IR-5
Incident Reporting	Incident Response Integrity	IR-6
Non-local Maintenance	Maintenance	MA-4(6)
Maintenance Personnel	Maintenance	MA-5
Timely Maintenance Media Protection	Maintenance	MA-6
Media Storage	Media Protection	MP-5
Media Sanitization & Maintenance	Media Protection	MP-6
Physical Access Authorizations	Physical & Environmental	PE-2
Physical Access Control	Physical & Environmental	PE-3
Access Control for Output Devices	Physical & Environmental	PE-5
Security Authorization Process	Program Management	PM-10
Vulnerability Scanning	Risk Assessment	RA-5
Application Partitioning	System and Communication Protection	SC-2
Information in Shared Resources	System and Communication Protection	SC-4
Boundary Protection	System and Communication Protection	SC-7
Transmission Confidentiality	System and Communication Protection	SC-8(1)
Cryptographic Protection	System and Communication Protection	SC-13
Collaborative Computing Devices	System and Communication Protection	SC-15
Protection of Information at Rest	System and Communication Protection	SC-28
Flaw Remediation	System & Information	SI-2
Malicious Code Protection	System & Information	SI-3
Information System Monitoring	System & Information	SI-4

Table 10. A List of Current Features of the Android Application Sandbox

Feature or Security Property
Application isolation of data and code execution from other apps
Robust implementations of security functionality (e.g., cryptography, permissions, and secure IPC)
Compiler-level Memory Management security technologies
An enablable encrypted file system
User-granted permissions for access to system features and user data
Application-defined permissions to control application data on a per-app basis
Secure credential authorizations (e.g., authorization tokens over user names/passwords)

III. Approach and Methodology

Modern threats and the lack of formally assured applications motivate the development of a new and assured container for Android. Isolation techniques can be used to protect host systems against potential threats. The two scenarios needing isolation were previously discussed in Chapter 2, and are the cases of malicious software on a trusted device and trusted software on a malicious device. The examination of Android development documentation indicates a form of malicious software isolation is targeted with the Android Application Sandbox, but the openly available documentation of its isolation mechanisms does not indicate that they are formally assured. This chapter discusses the planned methodology to design and implement a formally assured isolation tool on Android, which is referred to as Assured Android Execution Environments (A^2E^2). To propose a methodology for the development of an A^2E^2 container, this thesis describes and evaluates a method for selecting security properties, validating a set of properties defined in a specification, and evaluating existing technologies as verification avenues of a specification implementation.

III.1 Assured Android Execution Environments: Definition and Goals

The phrase "Assured Android Execution Environments" refers to a notional suite of security tools and applications that have been assured with validation and verification with formal methods to provide an Android execution space exhibiting specific security properties proposed herein. No specific tool or product is uniquely defined as A^2E^2. Rather A^2E^2 it is an umbrella term that covers any product having a validated specification and verified implementation for Android containment. This research results as a case study involving the development of a single container product to inform the general process and tools needed for A^2E^2.

One goal for A^2E^2 is the identification and adoption of an appropriate formal methods approach given the lack of a *de facto* standard. This method allows future A^2E^2 efforts to proceed in a standardized manner similar to a standard life cycle model for software (e.g., waterfall, agile, and/or spiral software development). Following a life cycle process identifies two additional goals for this research: (1) security properties used for A^2E^2 requirement definition that can be assured with formal methods, and (2) identification of applicable formal methods tools for specification validation and implementation verification. Furthermore, A^2E^2 efforts are not aimed at developing new formal method tools. Rather they leverage existing automated tools to enable rapid specification validation and implementation verification. These goals provide the starting vector in the approach to create realizable A^2E^2 products and methods.

III.2 Determining Methods to Achieve A^2E^2

Two main areas of research were initially identified when determining distinctive and practical for an A^2E^2 container. As previously stated there are no known list of proven security properties, so A^2E^2 needed to identify a set of security properties that could be leveraged in requirement definitions used in a specification (i.e., properties that can be validated in a specification and verified in an implementation). Once the set of security properties are established then a specification could be created which would be implemented in programming language.

However, a standardized approach in the application of using formal methods with traditional software development techniques was not identified in initial research for this topic. It was also identified that each security property within the specification would need individual assurance (i.e., if access control of encrypted data was a requirement, then both the access control and encryption would need specification validation and implementation verification). This process of assurance led to the identification of a new formal approach: the Formal Verification Cycle. The Formal Verification Cycle allows for assurance to be broken into two main stages: (1) validation of a specification that incorporates desired security properties and (2) verification of an implementation of the validated specification. The following sections detail an approach to identifying security properties, discussion of the two main steps for realizing a software specification and implementation (i.e., leveraging the Formal Verification Cycle), and the proposed three-stage approach used in this research.

Identifying Security Properties

The first step in achieving a software specification is evaluating, identifying, and defining system requirements. This is also true when seeking an Android container with desired set of security properties. A broad set of security properties will need to be found to begin a process of determining specific and applicable security properties. Those properties that can be assured, through formal methods, would be then form a set of general A^2E^2 security properties. Having an established and assured set of properties allows for future designs and the current design of a container to be a subset of properties needed to achieve the larger design security goals. However, there is not a widely accepted standard for defining what is or isn't an acceptable set of security properties in general terms or in specific terms for a software container. Thus, this research will need to determine a set of acceptable security properties and time permitting assure each property.

Standardizing A Formal Approach: the Formal Verification Cycle

As shown in Chapter 2 various methods and products fall within the class of formal method approaches. This previous work has inspired a way to standardize the method of approaching formal assurance, which is needed due to the lack of a widely accepted formal method approach/standard. Most individuals and organizations leveraging formal methods employ a single set of tools and an approach for assurance, but these approaches are commonly too specific for general development reuse.

Although some approaches to formal assurance can be replicated or duplicated for new designs, many instances of the techniques are specific to a particular implementation. This research demonstrates a new formal process based on work by the seL4 team be developed that aimed at leveraging existing tools. The team's process is modeled as the subprocesses of:

- Individual security property selection
- Security property theory validation to be captured in a specification
- Implementation of the design as source code and verification of the original specification

This process can be viewed as a three element cycle:

- Validation of the theoretical property (i.e., proving the theory)
- A validated system specification
- Use of the design specification to implement and verify a final software implementation (i.e., proving the implementation)

This three element approach is referred to hereafter as the "Formal Verification Cycle."

The first element of the cycle is broken out as the Theory wedge in Figure 4, and it consists of the steps of defining system properties that are later used with theorem proving tools to verify the design. The successful application of a formal method

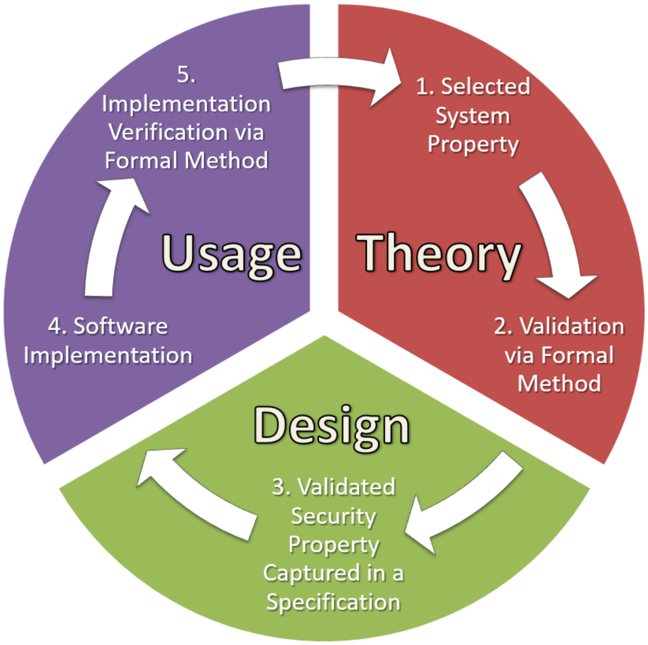

5. Implementation Verification via Formal Method
1. Selected System Property
Usage
Theory
2. Validation via Formal Method
4. Software Implementation
Design
3. Validated Security Property Captured in a Specification

Figure 4. The Formal Verification Cycle

(e.g., a proof assistant) results in validation of a selected security property. There is no product that is generated when successfully completing this process of validation (i.e., using a formal method like a proof assistant will show the theory as being true or false), but the validated property is now considered to be captured in a specification. Hence, the Design wedge in Figure 4 is a breakpoint for applied methods an tools in the Formal Verification Cycle, since a specification must have successful validation of a property being correct before an implementation is attempted.

Once the property has been validated, then second half of the Formal Verification Cycle would continue from the Usage wedge. The use of the validated specification allows for its implementation in a programming language of choice. Ideally a programming language should be selected that has a wide range of formal methods tool support.

Some programming languages have little to no support, which should drive implementation language decisions if ignoring other design or motivating factors. However, there might be external factors that drive a language selection (e.g., some languages might be chosen for existing work, targeted device supportability, supported tool chains, etc. that supersede the lack of formal verification tools), and if such the case were to occur new tools must be developed or modified to achieve implementation verification.

After language selection and an implementing of the specification are completed, then the resulting source code (or in some cases a form of the binary construction) can be verified with a formal method (e.g., models or abstract interpreters) to show the original security properties hold. Once a property has been successfully implemented, then all other properties could follow this cycle until all system requirements and security properties are implemented. Ultimately, this cycle illustrates how to verify an implementation of security properties built from the validated specification of the system. Many different tools and programming languages can be used, and this research does not propose the specific tools. However, future research may recommend tools based on best practices and ease of use.

The Formal Verification Cycle Identifies Three Distinct Issues

A specific approach is needed to define the methods, establish validity, and determine both applicable and non-applicable tools for use of A^2E^2 products. As previously mentioned the approach will start with the identification and selection of security properties that can meet the requirements of desired security features. Yet, an important discovery from using the Formal Verification Cycle is the establishment of two stages: security property theory validation with theorem proving, and implementation verification with model checking or abstract interpretation. This showed that A^2E^2 could be approached in three main stages to support the end task of implementing a container. The following section looks at each of these stages, and defines the approach in validation and implementation of an A^2E^2 container specification.

III.3 A^2E^2's Three-Stage Approach

As previously discussed, the approach of an A^2E^2 container design is in three major stages. The first stage identifies appropriate security property requirements, from an analysis of existing security frameworks and models, to propose a set of A^2E^2 security properties. The second stage focuses on a methods of validating security properties, with a theorem proving tool, to allow the creation of a validated specification. Lastly, the third stage evaluates a method of verifying software implementation with a symbolic execution tool (i.e., an abstract interpretation tool that leverages symbolic execution). The following sections detail the approaches for each of the three stages, and the resulting findings for each approach is detailed in Chapter 5.

Defining System Security Properties

To design and assure proper security properties for an A^2E^2 container a set of larger security features must be defined. The surveyed models represent best practices produced by subject matter expert committees and groups over time and are assumed here to have real world significance. Specifically, four models were surveyed, having various levels of implementation (i.e., conceptual design versus practical software implementation), and these were analyzed to create a final set of security properties.

The four models were not developed using with formal methods, but will act as the baseline for assurable security properties (i.e., properties that can be validated with a formal method). A consolidation will occur of the properties identified in the models to obtain a subset consisting of the properties potentially relevant to A^2E^2. After assurance of the properties then these are assumed to be valid for all subsequent A^2E^2 designs.

The security properties consolidated from the four models facilitate the definition of A^2E^2 System Security Properties. This process starts with the models discussed in Chapter 2, and mapping of the models' security properties to a single set for A^2E^2 (i.e., a non-duplicated set of properties will be defined if the four models show overlapping properties). This mapping starts at a general class of security properties , and applicable properties are selected based on the ability of the property to be implemented as source code (i.e., invalid and non-included security properties are commonly processes, activities, or in-person actions). Upon completion of the mapping activities a final set of security properties, for use across the class of A^2E^2 products, are recommended as a final of container security properties.

Security Property Validation

Once the selected set of security properties are chosen, then security property validation can be performed. It was decided that theorem proving would be leveraged to achieve property validation. As discussed in Chapter 2, the seL4 team leveraged Haskell (and proposed the tool of `Haskabelle`) to prototype and assure security properties. The tool of `Haskabelle` was selected to achieve the goal of tool reuse. The research aims to identify the method to setup, use, implement, and validate a Haskell-coded prototype. After the creation of a development environment, a simple use case of using `Haskabelle` showed a conversion of Haskell to Isar proof script. Proving the Isar script was accomplished with the Isabelle theorem proving tool. The tool and the process was selected since it includes a version of `Haskabelle` and the theorem proving tools needed for theory assurance of Isar script.

Implementation Verification

The final stage in this research is the verification of a programing language implementation. At this stage the proverbial rubber means the road, since the focus is real-world source code development that generates a usable software product. At the outset of the research it was unknown if any security property would be determined to be assured with theorem proving or with `Haskabelle`. The complexity of container development, lack of a final assured design, and unknown state of verification of the design theory prohibits a real-world solution for this research.

However, an approach was created that was independent of the theorem proving approach to showcase methods that are integrated into the workflow of a future A^2E^2 solution. It was determined that the Go language would be used based on the modeling of Docker, existing Go language developments (e.g., example containers, go mobile, and gollvm), and the symbolic execution tool `KLEE`.

Over the period of this research there was no identification of any formal method tools for the Go language. However, new advances in the generation of Golang source code and the tool "gollvm" showed that `LLVM-IR` could be generated from Go code. Thus, the intention of the research focused on method or modifications needed to show how `KLEE` could be integrated to evaluate Go-based `LLVM-IR`. If such a process was created, then it allows for the verification of a Go implemented container based on the existing example.

This research focused on first getting the Linux-based Go containers to compile (i.e., it was assumed if the container work on Linux then the `gomobile` tool could have a modified version built for Android). The second part focused on creating a `KLEE` development environment and showing an example implementation could be symbolically executed. Lastly, the final design of implementation verification focused on using `KLEE` to symbolic the Linux-based Go container implementation.

III.4 Assumptions and Limitations

When applying a mathematical model to a set of semantic actions there is a set of caveats and assumptions that are made. Due to the nature of logic proving, and mathematical representations thereof, there are issues that relate to how refined a representation can be. Additionally, when trying to prove security-related topics there needs to be a "build-up" approach (i.e., the smallest, most refined, unit is proven to be valid, and is proven up to less refined abstractions, which ultimately shows the whole approach is valid). This may not always be possible, and likely it is the case that a design is only be partially or specifically proven. This section seeks to highlight some of the relating issues for caveats and assumption of Android with respect to planned approach for A^2E^2.

Assumptions in A^2E^2

There are practical limits in the computation and verifiability for software products. Since A^2E^2 products are constructed as software, then they too inherit these limitations of computability. Therefore, all A^2E^2 proofs and approaches are bounded by the best practices or limitations of the respective tools used in design and implementation. Specific issues, if they arise, are annotated with respect to their identification within the results and conclusions portions of this research.

Additionally, a step toward realizable A^2E^2 instances is avoiding the re-validation or re-proving of results found in existing research. For example: if a tool or model exists, such as formal verification of the construction and execution of source code, then this research accepts the current issues and limitations for the defined tools and methods. It has been deemed out of scope to re-work efforts of existing products and tools, because this research shall assume existing approaches are valid to help ensure new forms of assurance. Additionally, existing tools integration or modification shall be considered valid if using such a tool as a module or if the tool is invoked to perform an analysis (i.e., the proof to assure and validate new tools or integrations thereof exceeds available resources of time and effort). If there are questionable issues in the quality or capabilities for chosen tools or methods, then these also are indicated in the results and conclusions section of this thesis.

Limitations in A^2E^2

One concern with formal verification is as it relates to "a formal supply chain." Ideally, total product assurance can only be realized when all portions of the system are formally assured with validation and verification via formal methods. This is not possible on Android, currently, since the hardware, operating system, nor its development tools are publicly formally assured. Knowing these facts, a truly formal solution cannot be presented in this research, since only a portion of the "formal supply chain" is being investigated. However, this research is looking to advance the general direction and topic of formal methods/formal assurance to enhance the security posture on Android.

Another major consideration for the research is the fact that assurance on Android is being approached as a high-level topic as opposed to specific implementation details with respect to the Android API and Framework. Given the complexity, immatureness, and magnitude of this work, the focus of the research is looking at broad classes of issues for assurance on Android. Thus, any proposed solutions are unable to delve into Android implementation specifics (e.g., coding API or compiler levels of issues). For instance: there are specific implementations in Android for encryption, but this research is not currently looking at the validity or assurance of this encryption as valid or not. At this point in A^2E^2 research, the focus is to enable and validate encryption as a security property, which is different than arguing the merits of Android's implementation of one standard versus another.

III.5 Expected Outcomes

At the outset of this research it was idealized that a singular and usable Assured Android container be developed or prototyped. Yet, with cursory research it was shown that this idealized state is impossible to be met for various reasons (e.g., extensive work for a single researcher, untested tools, lack of knowledge for functional programming verification,). The first and most important outcome for this research is a list of general security properties for the design and implementation of a secure design. Due to the lack of standards and accepted practices there is a need for a set of discovered or consensus-based security properties. This set of properties may be used in down-selections to pick specific and applicable A^2E^2 security properties. The findings from the security properties identification would then allow for the determination of tools and techniques of relating to both the proving of the formal specification and implementation. It is noted that there is no predicable amount of tools, techniques, or methods that are applicable to A^2E^2 work, so only the results to the analyzed and leveraged tool will be included.

IV. Results and Findings

The results and conclusions of the research herein directly relate to the outlined approach in Chapter 3. This three stage approach deals with: A^2E^2 security properties identification, validating security properties in a specification, and verifying a software implementation. The specific reference materials and stage-related content can be found in Chapters 2 and 3 respectively.

IV.1 Identified A^2E^2 Security Properties

A standardized set of security properties can be created from an evaluation of existing models and frameworks as discussed in Chapter 2. A key action is the consolidation of similar and overlapping properties that exist between the models and standards. These consolidated properties are proposed as the standard for A^2E^2 security properties.

Mapping Security Properties to a Single Model or Framework

Native and validated security properties are being identified for A^2E^2 to help simplify future assurance in software products. Each of the four models discussed in Chapter 2 are evaluated to support a consolidated list of native properties for A^2E^2, and the final identified set will be the result of consolidating the properties that may be validated. Validation of the properties is not executed at this stage, but a focus is given to the possibility of each property being implementable or not; for instance, it is important to have confidentiality from the CIA Triad, but confidentiality is not implementable where as cryptography is implementable and would achieve such a goal as confidentiality. Each framework will be reviewed for non-captured properties that will be appended to extend and create a final mapping of the four frameworks to a single set of security properties for A^2E^2. References for the contents and background of each framework were discussed in Chapter 2, thus only the minimal security requirements for each framework shall be considered for evaluation, and the analysis will start from the most general to most specific frameworks.

Selection and Incorporation of CIA Triad Properties

Generic security properties and themes are presented with the CIA Triad even with the extra features of authenticity and non-repudiation. On closer inspection the most enabling technology for CIA capabilities is cryptography, and it allows for the themes of Confidentiality and Integrity to be implemented. Nearly all security features could have a form of General Cryptography, thus it is the first security property to be considered for A^2E^2. No explicit details within CIA detail this general need for base mathematical capabilities, but it becomes apparent the impact cryptography has when looking at the technologies of one-way hash generation, one-time-pads, and cryptographic keys (both asymmetric and symmetric) and their use with ciphers. As such, core functions of cryptography enable larger security applications to be generated (e.g., key generation, encryption, decryption), and general cryptography stands as the base for many security functions and technologies.

Given the baseline of cryptography, the next focus in CIA examination looks to more specifics of the confidentiality of system data. It was observed that two main mechanisms that allow confidentiality to be achieved are data protection and access control. Very few security specification or designs could be proposed if they did not include mechanisms for data protection Elements and activities relating to protecting data are considered to be those that cannot be subverted or understood when the data is accessed (e.g., encryption protects data with mathematical operations that greatly increase the complexity and computation needed to interpret the protected data). Yet, encryption can also be thought of as a form of access control, since the only means of accessing encrypted information is with secret knowledge (e.g., a key) from a knowledgeable party (i.e., a person is granted permission and access to encrypted data only by being informed of the protected secret/means). Other, non-encryption, based mechanism exist for both data protection and access controls. Use of "whitelisted" security mechanisms provide authorized agents, or users, that are allowed access to non-encrypted data once validated. User namespaces and file system partitioning enable authorization of data access as other examples of access controls. Protection of data can be considered as a subset to access control properties, but both are considered as individual A^2E^2 security properties given the wide range of capabilities, methods, and protection mechanisms that exist for each category.

Integrity was the third property examined for security property identification for this research. Data integrity usually means that a specific set of data can be validated against a previous context (e.g., a one-way hash provides a fingerprint of sorts), and integrity implies that a given set of data is not changed while in rest or when transmitted. Like data protection the techniques of data integrity usually are implemented with general or base cryptographic methods. Yet, unlike data protection and access controls there are not guarantees regarding the privacy or access of other parties to the contents (i.e., integrity shows that data is not manipulated, and does not ensure access or protection mechanisms are circumvented).

The final included property for A^2E^2 inclusion was non-repudiated accounting. Specific capabilities of interest are for system logging and tracing capabilities (e.g., system logs). In modern computing it is critical to maintain and keep records of activities that occur on computing systems. More importantly is the ability to validate and ensure that records cannot be modified or falsified after occurring. Interesting developments in cryptocurrencies and block-chains show that such a ledger and journaling system can be applied in a wide range of applications, and as such their applicability to security are being explored to this day. An extension of such non-repudiation technologies could be a record and journaling of system resources and data as to support access control, data protection, and even data integrity applications.

There were two categories of the triad model that were not included for consideration of A^2E^2 security properties. The two properties that were not included were availability and authenticity. Availability was excluded for security properties, because the topic deals with the access to data and if at a given time such data can be accessed or not. There are no new security related concerns, since activities could be controlled or managed with access control mechanisms. It should be noted that if new security features are enabled with availability, then future iterations of A^2E^2 should re-evaluate its inclusion.

The second, non-included category of authenticity appears to largely focus on verification when used in higher level protocols or security applications. It is being argued that measures of authenticity are a mix between General Cryptography and Data Integrity implementations. For instance the act of authentication is some combination of validating a secret piece of information (e.g., a password that is then hashed) or ensuring that a set of data meets an original parameter (e.g., a signed message can be authenticated as a set of data that was configured or created with a mix of data integrity and data protection means). It is likely that a security implementation achieves authenticity goals via security properties previously discussed (i.e., access control, data protection, or data integrity), thus the explicit property of authentication is not used for A^2E^2.

At this point the initial A^2E^2 security properties from the CIA Triad categories can be seen in Table 11.

Table 11. The Initial Set of A^2E^2 Security Properties

CIA Triad-based Security Properties
General Cryptography
Access Control
Data Protection through Cryptographic Means
Data Integrity through Cryptographic Means
Non-repudiated Accounting

Selection and Incorporation of O-ISM3 Security Properties

O-ISM3 has core and recommended security processes that are collected in Table 5, but not all of these processes are realizable to be implemented in software. Effectively, 8 of the 16 essential processes can be removed since they are activities of organizational operations as opposed to implementable software capabilities (i.e., the operational processes that deal with the management of business activities cannot not be realized in a coded implementation). With the absence of administrative type-controls, the nine applicable set of security processes are available in Table 12.

Table 12. The Nine Essential O-ISM3 Processes

Process Type	Process Subcategory	Process Name
Generic		ISM Design and Evolution
Tactical		Define Security Targets & Security Objectives
Tactical		Service Level Management
Operational	Life Cycle Control	IT Managed Domain Patching
Operational	Life Cycle Control	Segmentation and Filtering Management
Operational	Life Cycle Control	Malware Protection Management
Operational	Access and Environmental Control	Access Control
Operational	Availability Control	Backup Management
Operational	Testing and Auditing	Information Quality and Compliance Assessment

The O-ISM3 provides no specific consensus of what is or isn't a secure system design, since the document mainly deals with the general ideas, concepts, and execution of security management. The model itself states there is "No One Solution Fits All" and every organization has a unique context and constraints. This implies that the O-ISM3 is used mainly for decision-making processes relating to security postures. Yet, this model prevails over the CIA Triad since it provides concrete security-related topics, and the O-ISM3 provides examples that have an emphasis on measurability of success via metrics. A set of themes of security features for the O-ISM3 was observed: access control and journaling of a system can be used to ensure the integrity of a specified design.

The first stage in identifying new security features focused on removal of duplication of topics. For instance the topic of access control was previously discussed with the CIA Triad.

Upon closer examination the processes of Segmentation and Filtering Management can be considered to as a topic of data integrity, since the description of the process details similar goals that are achieved in the property of data integrity. The four processes of (1) Define Security Targets & Security Objectives, (2) ISM Design and Evolution, (3) Service Level Management, and (4) Backup Management were also determined to fit within the existing A^2E^2 security properties. Specifically, the four process are looking at data integrity, since they are trying to ensure that a given choice of a process implementation meets an original target or design for a system/company.

The removal of the overlapping process provides three remaining processes: (1) IT Managed Domain Patching, (2) Malware Protection Management, and (3) Information Quality and Compliance Assessment. These bring a slight paradigm shift to the properties for A^2E^2, since these three processes attempt to accurately record and track known states with varying levels of information. The examination of each of these processes brings a slight modification of non-repudiated accounting is proposed: an addition of journaling. The terminology of accounting and journaling are very similar, but one main difference can be interpreted as active versus passive activities. In the case of journaling it should be considered as a passive, or background activity (e.g., logging), since it could be greatly desirable to have information of previous states or conditions to be recorded. However, the active components of accounting are the checks or analysis that would occur whether initiated automatically or manually. Therefore, the three processes show that there could be the case of passive tracking for a security system that may switch to an active accounting, but in both cases it is critical to ensure that both versions of record keeping ensure the property of non-repudiation. This analysis of the O-ISM3 now provides A^2E^2 Table 13 with the new set of proposed security.

Table 13. The O-ISM3 Amended Set of A^2E^2 Security Properties

CIA Triad-based Security Properties
General Cryptography
Access Control
Data Protection through Cryptographic Means
Data Integrity through Cryptographic Means
Non-repudiated Accounting & Journaling

Selection and Incorporation of DoD-based Properties

Not all of the minimum DoD-based controls are applicable to A^2E^2 security properties in a similar manner seen in the analysis of O-ISM3 processes. However, given the extensive and detailed list of controls in Table 9 an essay type analysis, as was completed with the O-ISM3, was not be accomplished. In place of an essay-style analysis the final set of the controls applicable to A^2E^2, with non-implementable controls removed from consideration, were generated by mapping the DFARS controls to an A^2E^2 security property. All mapped properties, mapped to A^2E^2 or non-implementable properties, can be seen Tables 14 and 15. The remaining, non-mapped controls are discussed for their merits of potential inclusion into A^2E^2's security properties in the following sections.

Four controls were found not to fit any of the existing categories based on the analysis of the CIA Triad and the O-ISM3.

The controls can be seen in Table 14 and are: (1) Information Flow Enforcement,

Table 14. Mapping DFARS Controls to Current A^2E^2 Properties

Control	Assigned Property
Account Management	Access Control
Access Enforcement	Access Control
Information Flow Enforcement	**NONE**
Least Privilege	Access Control
Unsuccessful Logon Attempts	Accounting & Journaling
Session Lock	Access Control
Remote Access	Access Control
Wireless Access	Access Control
Access Control for Mobile Devices	Access Control
Use of External Information Systems - AC-20(1)	**NONE**
Use of External Information Systems - AC-20(2)	**NONE**
Publicly Accessible Content	**NONE**
Security Awareness Training	Non-implementable
Audit Events	Accounting & Journaling
Content of Audit Records	Non-implementable
Audit Review, Analysis and Reporting	Non-implementable
Audit Reduction and Report Generation	Non-implementable
Timestamps	Accounting & Journaling
Protection of Audit Information	Data Integrity
Baseline Configuration	Accounting & Journaling
Configuration Settings	Accounting & Journaling
Least Functionality	Access Control
Information System Component Inventory	Non-implementable
Information System Backup	Data Integrity
Identification and Authentication (Organizational Users)	Access Control
Identifier Management	Access Control

(2) Use of External Information Systems - AC-20(1), (3) Use of External Information Systems - AC-20(2), and (4) Publicly Accessible Content. These controls can be divided up into two new A^2E^2 security properties, and the analysis is currently presented.

The first DFARS-based control of Information Flow Enforcement brings awareness of the ideas of data and execution flows at a program and system level, since there could be concerns with how data is being moved and how a process is executing within a system. For instance a Trojan obfuscates its malicious operations as benign

Table 15. Mapping DFARS Controls to Current A^2E^2 Properties (cont.)

Control	Assigned Property
Authenticator Management	Access Control
Incident Response Training	Non-implementable
Incident Handling	Non-implementable
Incident Monitoring	Accounting & Journaling
Incident Reporting	Non-implementable
Non-local Maintenance	Non-implementable
Maintenance Personnel	Non-implementable
Timely Maintenance Media Protection	Non-implementable
Media Storage	Non-implementable
Media Sanitization & Maintenance	Data Protection
Physical Access Authorizations	Access Control
Physical Access Control	Access Control
Access Control for Output Devices	Access Control
Security Authorization Process	Access Control
Vulnerability Scanning	Accounting & Journaling
Application Partitioning	Access Control
Information in Shared Resources	Access Control
Boundary Protection	Access Control
Transmission Confidentiality	General Cryptography
Cryptographic Protection	General Cryptography
Collaborative Computing Devices	Non-implementable
Protection of Information at Rest	Data Protection
Flaw Remediation	Data Integrity
Malicious Code Protection	Access Control
Information System Monitoring	Accounting & Journaling

activities, but an execution flow control could possible prohibit or prevent such actions to occur. Additionally, if data is being moved, located, or even being transmitted in a certain fashion this could lead to data leakage or spillage that would not normally be covered by access controls. Therefore a new security property of Data and Execution Flows is proposed as to help mitigate malicious activities and enforce a means of controlling and enforcing both activities.

The next security property being proposed deals with the conditions of external versus internal security conditions. Effectively, the controls of Use of External Information Systems and Publicly Accessible Content showcase a special case of access controls that deal with issues external to a system. In essence, a security concern may stem from interactions external to the designed system and such input and output would need to be developed and designed in a different manner than fully realizable, and internal, solutions. An example external communications could that of Application Programming Interfaces, networking protocols and communications, and even concurrent or shared memory for a design. In these cases special care may need to be given that extend beyond capabilities of data protection, data integrity, or access controls (e.g., a web server or web service is usually designed to be open and accessible, which seems the opposite of security). Likely the use of current A^2E^2 properties are leveraged to implement and build up to a specific design, but given the complexity and special nature of external impacts (i.e., effects that are out of control of a current design) a new category is proposed.

The consolidated set of proposed A^2E^2 security properties are seen in Table 16.

Table 16. The DoD Amended Set of A^2E^2 Security Properties

DoD-based Security Properties
General Cryptography
Access Control
Data Protection through Cryptographic Means
Data Integrity through Cryptographic Means
Non-repudiated Accounting & Journaling
Data and Execution Flows
External Communications Security

Selection and Incorporation of Android Properties

The easiest selections of properties to evaluate for A^2E^2 are those that already exist in Android. To reiterate from Chapter 2, the Android Application Sandbox has the following features: data management (internal, external, or content providers), permission configuration for device assets (e.g., the camera), permissions for networking, input validation, cryptography, inter-process communication (network sockets, shared files, intents, binders, or messenger), dynamically code loading, and Dalvik virtual machine settings. However, at this point the security of the Android Sandbox does not provide any new category of A^2E^2 security properties, because each of the security features could map to at least one existing category. This does not imply that the current list of A^2E^2 properties are fully encompassing or completed, but it perhaps showcases how the analysis of several models can help identify core properties that can be leveraged across multiple designs.

The Proposed Security Properties for A^2E^2

Given the research into security properties from the previous models and system, and seeking the establishment of system design requirements, this research proposes the following categories of security properties for system design: General Cryptography, Access Control, Data Protection through Cryptographic Means, Data Integrity through Cryptographic Means, Non-repudiated Accounting & Journaling, Data and Execution Flows, and External Communications Security. These are in fact the final properties as proposed in Table 16. As oft mentioned, this list does not preclude the addition of new properties, so perhaps a most accurate description is this set of security properties being the version of a continually updated and analyzed framework. The intention for this design is to combine and consolidated like properties as to enable full system security, since it should be noted that these properties could be extended and be applied beyond the usage of assured application isolation. This can be important when approaching a design problem from the perspective of a the Formal Verification Cycle. Having examined several different approaches to security properties and controls, this final and consolidated distillation forms the basis for A^2E^2 security. Hopefully, this encourages and enable resource of proven security properties; such a practice could be thought of as Proof Designs Once, Implement Everywhere (...which also needs assurance via proof).

Recommended Security Properties for an A^2E^2 Container

The final consideration for security property design and requirements is given to the implementation of an A^2E^2 container with the newly defined security properties. At this time the properties are a recommended best practice, since no explicit property was defined and formally verified (i.e., these properties would need explicit implementations and verification to become a known and assured security method). An issue discovered by this research is the "chicken and egg" type problem, since the desired security properties must be defined and proven but at the outset of the research no known tools or techniques were known to verify such security properties. Assuming that the development of an A^2E^2 container were to be completed, then the proposed security properties would also become validated and assured. As it stands not all properties proposed for A^2E^2 are needed to implement a Android container, but the point is conceded that each property could be applicable to a container. Thus, in terms of this research an A^2E^2 container is being sought that isolates the partitioned file directory and execution of processes in a single environment, so to help drive and define the research for the methods identification for theory and implementation assurance. In this case the security properties of A^2E^2 that would likely need implementation are: General Cryptography, Access Control, and Data Integrity. Simply: the first generation container should be able to isolate its process execution in a container, which is able to be saved in a state that is able to be paused, stopped, or resumed. This first generation container is similar to Chroot, since applications would run within their own space on an Android device, and any enhanced features (e.g., communication control with Android and other containers, activity monitoring and logging, encrypted containers and asymmetric key infrastructures) is left for future research.

IV.2 Specification Assurance Findings

This section describes an applied approach to formal validation of security property when prototyped with a functional programming language. The goal of validation was attempted with `Haskabelle` and `Isabelle`. This section highlights establishing a validation environment, prototyping an example Haskell program, conversion to Isar for `Isabelle` with the `Haskabelle` tool, and an execution of the proof assistant to show the validity of the converted prototype.

Establishing a Validation Environment

Using the seL4 Development Environment with Docker

The first step in specification validation focused on the creation of a stable validation environment. The seL4 team published and made freely available a Docker-based assurance and proving environment, which was discussed in Chapter 2. Instructions provided on the team's development blog were leveraged on a new installation of Ubuntu 17.04. The directions were straight forward, and it was verified that the stated system requirements (e.g., RAM and storage space) were the minimum required requirements that allow successful configuration and execution of the environment (i.e., provided system resources under the recommended amount crashed when executing Isabelle proofs). No further instructions or use of this environment was analyzed once the main seL4 Isabelle proofs successfully executed (i.e., the environment is offered to re-validate or extend the work of seL4, but this line of research and usage was not needed for A^2E^2). This provisioning effort was captured in a BASH script called "cl4c" (or "CLACK"), which automates the blog instructions to setup the Docker-based environment.

Having the environment installed, and the cl4c provisioning script available, it was determined that this environment will be used for Haskell to Isabelle validation efforts. However, this development environment was deemed to be too over-featured (i.e., the environment includes all needed tools for seL4 development from source code), and did not directly support deployment on Android. Specifically, the environment lacks needed tools for Android implementations (e.g., Java or Bionic-C based implementations). Thus, a simpler means to provision and deploy an Isabelle environment for A^2E^2) was identified.

Isabelle 2017 and Docker

It was decided to discover a solitary environment for `Isabelle` given the complex system requirements and non-needed features in the seL4 environment. A proving and validation environment featuring `Isabelle` within a Docker container was found. A Dockerfile (i.e., a Docker provisioning and configuration script) was found on Docker Hub, which is a freely hosted website with a collection of Docker scripts and containers by Docker Inc. The Dockerfile runs on a computer with Docker installed, but it was identified that this Dockerfile needs a pre-downloaded copy of the 2017 edition of `Isabelle`. This copy of the tool is located on the main `Isabelle` website as a single archive. Once the Dockerfile and the `Isabelle` archive are contained in the same directory, then the Docker container can be built and automatically provisioned with following command: '*docker build .*'.

After the Docker container is built the created images can be seen with the command: '$docker\ images\ -a$'. To establish a baseline and restorable image for the container a tag can be used. This tag can be set when using the identification code with the following command: '$docker\ tag\ (ImageID)\ (Repository : Tag)$' (e.g.,

‘$docker\ tag\ 123412341234\ Haskabelle2015 : initial$’). With the Docker container built and the tag established, which allows the environment to be restored to this initial state, then container can be invoked in an interactive mode via the command: ‘$docker\ run\ -it\ Haskabelle2015 : initial\ /bin/bash$’.

This simplified Docker container exists and enables `Isabelle` use with having the latest 2017 edition. This container holders the proving and validation tools similar to the seL4 team environment, but this new container does not have any undesired or unneeded files (e.g., the seL4 source files). This environment allows for validation of Isar script files once the files are moved into the container. However, this variant of an Isabelle environment does not achieve an easier method of converting Haskell into proof script automatically, since it was discovered that `Haskabelle` is not included in the 2017 edition.

Isabelle 2015 with Haskabelle and Docker

`Haskabelle` is a tool that automates the process of converting Haskell source code to Isar proof script. This tool was seen as useful for A^2E^2 since it aims to reduce the workload, time, and tools needed to generate proof script from a Haskell prototype. Given the advantages of `Haskabelle` a final environment was sought to provide the tool with `Isabelle` to provide a development and validation environment for A^2E^2.

`Haskabelle` does not have a sole website that acts as the central repository for information or executable versions. Two resources found were on the main `Isabelle` website and a GitHub.com mirror of the source code [82][83][84][84]. It was determined the main distribution method for `Haskabelle` was its inclusion into the main `Isabelle` distribution/archive. However, the tool was not present in the 2017 edition, which as confirmed with the previous Docker container setup. This absence of Haskabelle is also true in the 2016 edition, and the last updated version that includes Haskabelle is the 2015 edition of Isabelle.

Hence, a Dockerfile was extended from the Dockerfile for Isabelle 2017 to instead install and provision the 2015 version. The Dockerfile for the Isabelle 2015 edition was ensured to have the proper dependencies and auto-downloads the needed archives to automate the configuration and provisioning of a Docker image. This new container installation includes both Isabelle and Haskabelle in a single container. An additional feature was the creation of a work space, to store developed and validation files for Isabelle, and this folder is located in the root directory with the name Isabelle_Workspace/. The final action for this environment was to execute the Docker commands to set image tags, which were followed as detailed in the previous section.

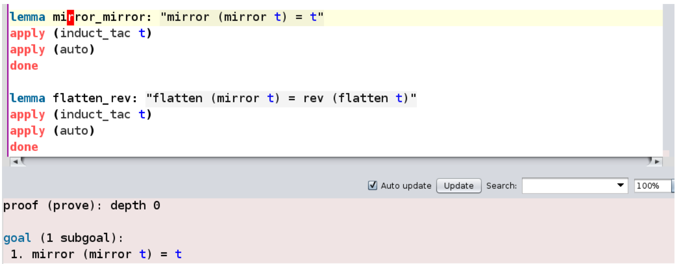
Applied Tools and Methods for Security Property Validation

Using Haskabelle with Isabelle

The final environment created allowed the use of both Isabelle or Haskabelle and Isabelle is used. It should be noted that this environment is configured for running command line-based invocations. If the graphical version of Isabelle is required then it should be installed outside of the Docker container (e.g., another Virtual Machine or directly on the Host Machine). When using the container, a common issue arises with the import and export of development files can be a problem. As a recommendation: an external Git repository works well to push and pull data in or out of the container. Yet, another solution, with Docker, focuses on mounting a portion of the host machine space into the container, but this method was not leveraged in this research. Once the Haskell source has been moved into the container, then Isabelle can the Haskabelle generated Isar files.

At this time no specific security properties for A^2E^2 were modeled or implemented as a Haskell prototype. However, a simple Haskell program was used to explore and evaluate Haskabelle with the prototype of a Binary Tree via the Haskell BTree module. A tutorial by Seipp was found and it shows a methodology of using Haskabelle to generate the the Isar theory files. Additionally, the tutorial shows how to place in the desired steps and constraints for proving the prototyped code into the generated Isar source.

Specifically, the tutorial takes the actions of (1) creates a base Binary Tree in Haskell (e.g., a BTree.hs source file), (2) leverages Haskabelle to generate the Isar theory files (e.g., BTree.thy and the source file Prelude.thy for Haskell definitions in Isabelle), (3) by-hand addition of lemmas into the generated Binary Tree theory file (e.g., BTree.thy), and (4) Isabelle execution of the files to validate the Haskell prototype in its Isar represented form. Completing these actions and executing the resulting theory files with the command line version of Isabelle can be non-intuitive for analysis. Another method used for evaluating the execution of Isabelle is the visual inspection of the Isar files via the GUI version of the tool. Figure 5 through Figure 12 show screenshots of the final versions of the tutorials code with the respective output of line-by-line processing.

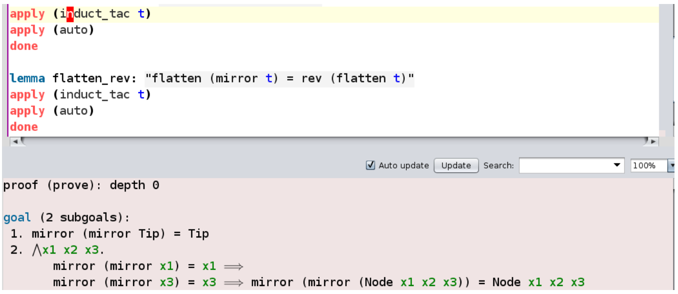


```
lemma mirror_mirror: "mirror (mirror t) = t"
apply (induct_tac t)
apply (auto)
done

lemma flatten_rev: "flatten (mirror t) = rev (flatten t)"
apply (induct_tac t)
apply (auto)
done
```

```
proof (prove): depth 0

goal (1 subgoal):
 1. mirror (mirror t) = t
```

Figure 5. Added Lemma 1 Initialization in Isabelle

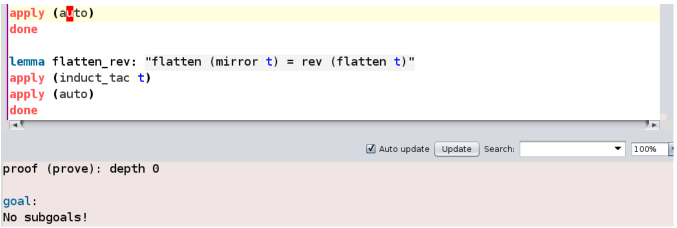
The output of Isabelle is not immediately apparent for the tool's results. There is no explicit pass or fail indication on screen when processing a theory file. There



```
apply (induct_tac t)
apply (auto)
done

lemma flatten_rev: "flatten (mirror t) = rev (flatten t)"
apply (induct_tac t)
apply (auto)
done
```

```
proof (prove): depth 0

goal (2 subgoals):
 1. mirror (mirror Tip) = Tip
 2. ⋀x1 x2 x3.
       mirror (mirror x1) = x1 ⟹
       mirror (mirror x3) = x3 ⟹ mirror (mirror (Node x1 x2 x3)) = Node x1 x2 x3
```

Figure 6. Added Lemma 1 Induction Strategy Applied in Isabelle

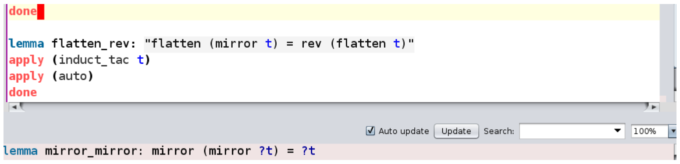


```
apply (auto)
done

lemma flatten_rev: "flatten (mirror t) = rev (flatten t)"
apply (induct_tac t)
apply (auto)
done
```

```
proof (prove): depth 0

goal:
No subgoals!
```

Figure 7. Added Lemma 1 Induction Strategy Finished in Isabelle

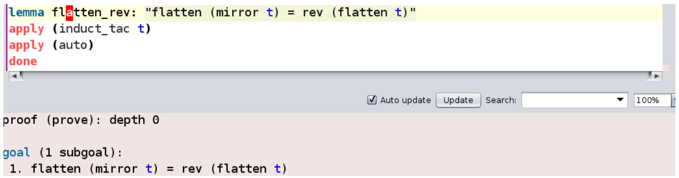
are certain means to identify successful execution for a given source file. Specifically, the documentation for `Isabelle` states than if the output shows $?x = ?x$ then the tool is able to instantiate the provided lemmas arbitrarily which means the lemmas are valid [85]. The final `Isabelle` output in Figure 8 and Figure 12 shows the case that $?t = ?t$ when applying a flatten inductive strategy to the Binary Tree. This means the the tutorial `Haskabelle` output (e.g., BTree.thy) with the inserted proof logic for the lemmas were successfully executed in `Isabelle`. This process could be extended for other cases of prototyped Haskell code. This example shows a single implementation, but it is conceivable that A^2E^2 security properties could be proven



```
done

lemma flatten_rev: "flatten (mirror t) = rev (flatten t)"
apply (induct_tac t)
apply (auto)
done
```

```
lemma mirror_mirror: mirror (mirror ?t) = ?t
```

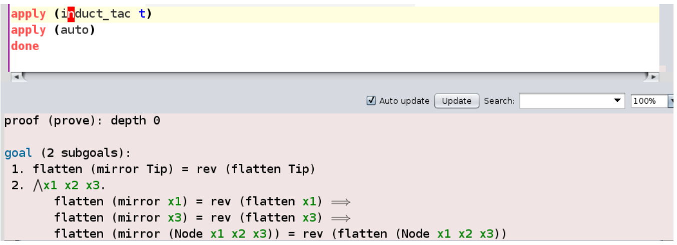
Figure 8. Added Lemma 1 Work Finished in Isabelle



```
lemma flatten_rev: "flatten (mirror t) = rev (flatten t)"
apply (induct_tac t)
apply (auto)
done
```

```
proof (prove): depth 0

goal (1 subgoal):
 1. flatten (mirror t) = rev (flatten t)
```

Figure 9. Added Lemma 2 Initialization in Isabelle



```
apply (induct_tac t)
apply (auto)
done
```

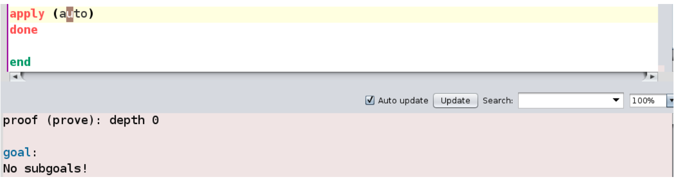
```
proof (prove): depth 0

goal (2 subgoals):
 1. flatten (mirror Tip) = rev (flatten Tip)
 2. ⋀x1 x2 x3.
       flatten (mirror x1) = rev (flatten x1) ⟹
       flatten (mirror x3) = rev (flatten x3) ⟹
       flatten (mirror (Node x1 x2 x3)) = rev (flatten (Node x1 x2 x3))
```

Figure 10. Added Lemma 2 Induction Strategy Applied in Isabelle

or disproven when following this outlined validation methodology.

Current Issues in Leveraging `Haskabelle` for Automated Theorem Proving

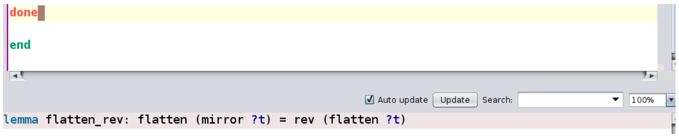
The tutorial provided a process to validate Haskell prototypes with the steps of: (1) generating a prototyped feature in Haskell, (2) converting the code to Isar with `Haskabelle`, (3) adding the appropriate logic and theory lemmas into the Isar, and (4) leveraging `Isabelle` for validation of the source code. These methodology proves and validates a Haskell prototype can be accurately modeled in a functional programming language. The process of leveraging `Haskabelle` will generate an Isar representation. Yet, this method is only a partial automation, since it cannot



```
apply (auto)
done

end
```

```
proof (prove): depth 0

goal:
No subgoals!
```

Figure 11. Added Lemma 2 Induction Strategy Finished in Isabelle



```
done

end
```

```
lemma flatten_rev: flatten (mirror ?t) = rev (flatten ?t)
```

Figure 12. Added Lemma 2 Work Finished in Isabelle

generate the semantics needed for automated theorem and lemma implementation, which was seen with a non-trivial process of adding the appropriate logic for ensuring a correct design. The tool stops short of fully automating a solution, since it has no means of including or generating the needed Isar proof logic.

IV.3 Implementation Verification Findings

A Model System and Approach: Docker

Once the first two stages of the Formal Verification Life Cycle are completed (i.e., security properties are identified and each are validated) then a specification can be developed and implemented in a programming language. The research assumed a specification for a container was valid, because it allowed the discovery and identification of formal methods for implementation verification. Hence, this research looked at imperative programming languages, implementation verification tools, and operating systems that would support Android container development.

Similar capabilities exist within Linux android operating systems, since these systems are based off of the Linux kernel. It was determined that containerization on one OS would be equivalent to the other for a real world implementation. The Docker tool was modeled to support the realization a Android-based containment implementation, since Docker executes on Linux kernel as a container environment/tool. Docker has used the Go programming language to implement container mechanics (i.e., Docker was programmed in Golang and leverages Linux kernel features and function calls to achieve namespace-based isolation), and the programming language has applicable

Android application tool-chains for development. Therefore this research sought a method of containment with Golang, since an implementation could be theoretically ported to Android.

Determination of Tools and Approaches for an A^2E^2 Container Implementation

It was assumed for this research if a Linux software container could be constructed, then a conversion of the container to an Android version of the container is possible. Software development tools exist for both Linux android, so the only undetermined tool(s) relate to formal methods supporting implementation verification. The driving factor for formal method selection is based upon the implementation programming language, which was determined with the modeling of Docker. Techniques and tools were reviewed to determine which of the three main approaches of formal methods could be applied to Go-based code. The selected the formal method technique to be leveraged was abstract interpretation, and it was determined that symbolic execution offers a new verification method for Go. The following sections discuss the selected methods and tools needed to generate and verify a Golang-based software container.

A Go Implementation of a Linux Container

As discussed in Chapter 2 there are multiple examples of Go containers online. It was determined that Rice's container implementation was going to be used in this research after reviewing several different implementations of Go-based containers. The code offered a simple container that was under one hundred lines of Go code, and there are several sources documenting its implementation and development. Rice's code was tested and confirmed to execute on Ubuntu Linux 16.04 based on the GitHub hosted source file. This Linux container built in Go was not validated to be an A^2E^2

specification, but the container shows how an implementation could be created and subsequently verified. Given the simplicity of the design it was determined that security property validation could be extended or reuse Rice's implementation in capturing a valid A^2E^2 specification.

Verifying Program Correctness with Symbolic Execution

In order to assure an implementation, via verification, abstract interpretation of Go executables was researched. There are various formal methods that satisfy abstract interpretation, but it was decided to leverage symbolic execution for a Go implementation. As discussed in Chapter 2 there are many benefits to leveraging symbolic execution, but the driving factor for this decision was the discovered tool KLEE.

KLEE presented a challenging approach, since it does not have a presence with native, C-based, Android applications nor does it have an implementation for Go-based code. However, a recent tool, gollvm, had been introduced into the Go development pantheon, and it is based on the LLVM compiler infrastructure that KLEE uses. It was decided to that a Go implementation could be verified by using KLEE's symbolic execution if it were able to process the outputs provided by gollvm.

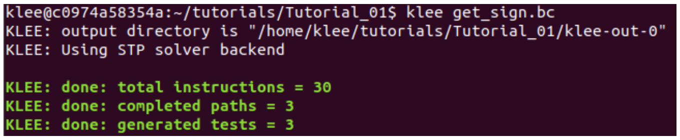
Findings in Applied Implementation Verification Methods

KLEE Analysis and Evaluation

The final effort for the research focused on developing the techniques needed to leverage KLEE with the Go container implementation. This attempt is the known attempt of creating a Go abstract interpreters with a symbolic execution tool/framework. The selection of KLEE was motivated by the tool's use with C/C++ source code, since the source code and libraries are can be invoked by Go program. Another motivating factor is leveraging KLEE was based on the large amount of documentation and tutorials for using the tool.

Before analysis and use of the tool started a final action of selecting and building a development environment was completed. The KLEE can be built from the source code for a respective environment (e.g., bare metal host machine, development environment virtual machine, etc.), or the tool can be leveraged with a provided Docker environment. The Docker development environment was selected, since issues relating to non-resolved dependencies were discovered when building the tool from source code. Specifically, the source code building failed due to a dependency upon the LLVM project's code base (that is considerably large), and the specific issue causing compilation errors was not discovered. Hence, the work and source code were simply executed within the KLEE-provided Docker container, and the development files were moved into the container for analysis.

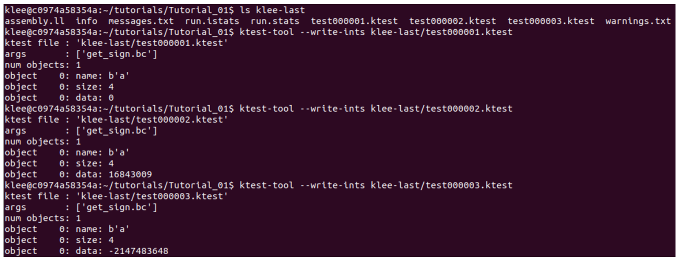
The main method to ensuring the proper setup for the KLEE environment was accomplished by following the tutorials provided on the main *klee.github.io* website. The base example, or the "Hello World" for KLEE, is programmed in C/C++ and was executed to validate proper setup and configuration of Docker development environment. This example leverages the primary KLEE function call (i.e., klee_make_symbolic()) in a simple program comprised of a series of if-else statements, which test an integer to be positive, negative, or zero. Several resulting analysis and runtime files were created with the source code when it is compiled and executed. Most importantly there are methods to generate the LLVM Intermediate Representation of the source code via the LLVM Clang compiler. The LLVM-IR can be generated from the source code when it is compiled with Clang. The compilation options of "-I" flag and the setting of "-emit-llvm" result in the generation of bitcode as a ".bc" file. The bitcode file was directly executed by KLEE, and the output of the tool is seen in Figure 13.



```
klee@c0974a58354a:~/tutorials/Tutorial_01$ klee get_sign.bc
KLEE: output directory is "/home/klee/tutorials/Tutorial_01/klee-out-0"
KLEE: Using STP solver backend

KLEE: done: total instructions = 30
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
```

Figure 13. Example KLEE Execution of LLVM-IR bitcode

The symbolic execution of the source code is automatically generated, and the used test cases and outputs were evaluated. Figure 14 shows the simple test cases and the results from KLEE (i.e, the results of the test for positive, zero, or a negative number).



```
klee@c0974a58354a:~/tutorials/Tutorial_01$ ls klee-last
assembly.ll  info  messages.txt  run.istats  run.stats  test000001.ktest  test000002.ktest  test000003.ktest  warnings.txt
klee@c0974a58354a:~/tutorials/Tutorial_01$ ktest-tool --write-ints klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['get_sign.bc']
num objects: 1
object    0: name: b'a'
object    0: size: 4
object    0: data: 0
klee@c0974a58354a:~/tutorials/Tutorial_01$ ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['get_sign.bc']
num objects: 1
object    0: name: b'a'
object    0: size: 4
object    0: data: 16843009
klee@c0974a58354a:~/tutorials/Tutorial_01$ ktest-tool --write-ints klee-last/test000003.ktest
ktest file : 'klee-last/test000003.ktest'
args       : ['get_sign.bc']
num objects: 1
object    0: name: b'a'
object    0: size: 4
object    0: data: -2147483648
```

Figure 14. Example Evaluation of KLEE Execution Output

At this point it was determined that KLEE was able to be leveraged as an abstract Interpreter with the used of symbolic execution.

Attempted Integration of KLEE via gollvm

The remaining task in applying KLEE to the Go-based container was two folded: (1) discover a means of generating Go-based LLVM-IR, and (2) integrate the KLEE tests into the base Go code. In the attempt to generate LLVM-IR, two discovered Go/LLVM tools were found. The tools llgo and gollvm were evaluated. The llgo tool was one of the first attempts seeking to integrate LLVM and Go, since it sought to generate Go executables with the LLVM framework. The tool and its sources are available on both GitHub.com and are included in the main LLVM project. However, the tool did not successfully compile or run per the provided instructions during this research.

The gollvm tool was found under active development, and is integrated into the official Go language experimental source file repository. This tool was successfully used alongside the full LLVM framework, and compiled an example Go program's source code to the LLVM-IR bitcode. However, a difficulty identified in this approach and tool was the fact that there is no easily development environment provisioning script. There were many issues building the tool from source with respect to the LLVM source. However, two stable git commits were discovered for both LLVM and gollvm that created a stable gollvm development environment. A set of a BASH provisioning and installation scripts were generated to replicate this development environment, and the scripts can be found in in Appendix C.1 and C.2.

At this point LLVM and gollvm were executed, but only in a manner that is currently independent of KLEE. It was successfully determined that the gollvm tool provides a sub-tool called llvm-goparse. This new tool was leveraged to generate the LLVM-IR with the same Clang command line flag of $-dump-ir$. An example program, provided by the gollvm team, was used and its use with KLEE can be found in Appendix C.3 through Appendix C.5.

The complete integration of KLEE's C/C++ based calls for gollvm compilable Go code was not completed. Thus, the use of KLEE to verify a Go-based container is unknown to being a possible formal method or not. Also, considerable effort remains to a single, joint KLEE and gollvm development environment. One finding for these tools was the fact that gollvm is experimental in nature, and its code-base seems to be unstable with the main LLVM framework. Additionally, The efforts attempting integrate gollvm into the KLEE-provided Docker image failed,

V. Conclusions and Recommendations

This chapter summarizes the state of A^2E^2 research and presents both lessons learned and recommended future work. A list of security properties relevant to A^2E^2 was generated (Chapters 3 and 4), and these properties can be used to direct future efforts with the Formal Verification Cycle. It remains to assure the security properties and to complete the validation and verification of a specification and implementation of an A^2E^2 container.

V.1 Three Stage Approach Overview and Research Contributions

This thesis focuses on a three stage approach to formal assurance on Android. The first stage identifies and defines security properties, the second identifies and applies tools for property validation in a specification, and the final evaluates new implementation verification methods for a Go language-based container. Each of the stages provided new insights for A^2E^2 formal assurance, and this section reviews the general findings for each.

Security Properties for Assured Android Container Specifications

There is not a *de facto* standard set of security properties for use in a formally verified specification. The consolidation of security properties led to the discovery of a "the chicken or the egg" problem: the final properties proposed need to be validated with a formal tools and methods (e.g., proof assistants, model checkers, abstract interpreters), but a final set of tools and methods could not be executed without verifying applicable tools (i.e., feedback from the second and third stages of research identified some tools and methods and how to apply them to validating and verifying a specification). Additionally, it was determined the complexity for full validation of each property was too extensive to be fully realized in this research. The security properties should be viewed as best practices, since they have yet to be validated with formal assurance methods. Once a final tool and approach are selected, the proposed properties can be validated for an Android specification. When the properties are assured then the result would be an *a la carte* menu for core security properties and features selection.

Lessons Learned with Former Specification Validation Approaches and Tools

The approaches examined in the first stage of the research illustrate methods that can be used for reasoning with respect to validation of a theoretical design of a specification. That being said, there were multiple lessons learned with the researched methods used by the seL4 to changes over the past eight years. A large impact to functional prototyping with Haskell has affected the `Haskabelle` tool and presumably seL4. The Haskell language, in 2014, had a fundamental change for their monad implementations as discussed in Chapter 2. Given the extensive use of monads by the `Haskabelle` tool it is no longer compatible with modern `Isabelle` versions, and until the underlying code and functionality of the tool are updated to the new monad standard, only the old versions will work. This does not necessarily prevent assured development, but it brings into question whether these tools can be used for sustainable or future solutions.

Two main discoveries were with the applied sections of this research. The first discovery was the identification that `Haskabelle` was considered deprecated in 2015 by the `Isabelle` team. The second discovery was that the former methods of the seL4 team would not be able to be reused for this research (i.e., the tools were not discovered for Haskell to Isar conversion and are presumed to be too specific for seL4 for general assurance work). As such, a Docker environment was created to support older versions of Isabelle and Haskabelle illustrating that former methods are still useful in specification assurance. This environment was validated with a Haskell prototype that was processed and verified by Haskabelle and Isabelle. The set of provisioning scripts used to setup and configure the Docker environment can be found included in Appendix C.1 and C.2.

Formally Assured Go Language Applications

The final stage of research focused on the identification of new methods of implementation verification for Go language applications. This approach was patterned on Docker's implementation, since it provides containment method the Linux kernel. Applications created with the Go language may offer a new approach for source code implementation that is portable to Android (e.g., the two main approaches for Android are with Java-based or native code-based programming). It is unclear whether the Go language container can be directly executed on Android (in particular, issues remain with OS permissions and Android tool-chain compatibility with the Gomobile suite). Cross-compiling the Go source code to Android was beyond the scope of this effort.

Advances in the gollvm tool enable the building of Go code within the LLVM compiler infrastructure. A compatible gollvm and LLVM versions, was found, and Docker was used to create a stable development environment. This process allowed a repeatable and stable build process. The Docker container allows for example Go-based Linux container to be built with LLVM into the LLVM-IR, which was needed as input for KLEE's symbolic engine.

KLEE is based upon the C language, and there are no Go language bindings for KLEE. A focus of the third stage was attempting the integration of Go with C functions and libraries. Ideally the integration would allow Go-based implementations to invoke `KLEE`'s C/C++ functionality. It remains to create Go bindings for `KLEE`. The Go container was built to LLVM-IR, and the `KLEE` examples in C/C++ were successfully executed.

An Initial A^2E^2 Framework

Each of the three stages of research made progress. These findings provide a formal framework for the selection of formal assurance properties, which are validated and verified with theorem proving and by means of symbolic execution. Future work needs to be assured, and the reusable `Haskabelle/Isabelle` environment can be leveraged in future work. Additionally, this research has shown a potential path forward in the Go programming language if symbolic execution was applied to show implementation verification. This initial framework shows applied tools and formal methods used in formal assurance (i.e., validation and verification) of A^2E^2. Other tools may exist that are easier to use or more stable and that can sustain ongoing assured development.

V.2 Other Findings

In this section, the topics of tool reusage, cautions on immature programming languages, identification of issues with past approaches, and cursory research into new and upcoming formal method techniques are presented.

Benefits of Formal Method Tool Reusage

There are benefits to using legacy tools even if they are deprecated or abandoned. Older versions of tools sometimes still work, and some methods presented by past research are still applicable to modern assurance attempts. Specifically, in appropriate circumstances, legacy tools in container-based development environments are preferable to tool versions that support host machine-based installations.

However, for a production environment up-to-date tools should be used. In particular, when leveraging Haskell tools such as `Haskabelle` are deprecated and not recommended. The development of new tools can be quite intensive in total man-hours, but such should be considered, especially for collaborative efforts. Furthermore, tradeoffs exist between formal method tools, so another consideration should be in matching new efforts to existing work a respective field of study. For examples, some organizations could focus on model checking for assurance, as opposed to theorem proving, so new work may be able those efforts instead.

Difficulties in Formal Verification of Immature Language

The Go language was released in 2009 and has gained popularity since the release, but as of this writing it is an immature language. Many existing and established languages have many tools that have accumulated since 1976 to support formal assurance. Yet, the Go language was created recently, so it has not had the benefit of formal assurance tools. As of this writing there is no evidence of major efforts toward formal verification for Go. New tools must be implemented for Go such as a Haskell to Go tool for prototyping, and completed Go bindings for `KLEE`. Additionally, no formally verified Go compiler is currently available. Hence, additional tools are needed to interpret and verify Go compiled binaries, and an effort to build a formally verified Go compiler must be completed.

New Developments of Formal Method Approaches

Non-supported, deprecated, or abandonware is not uncommon with many open source and academic research tools. Knowing the current state of `Haskabelle`, stemming from the changes in Haskell, there might be motivation reevaluate tools choice for formal assurance. The current method of leveraging the Formal Verification Cycle could be kept as a model, and it could be leveraged and executed with new tools. A first order examination of current research shows a new formal proof assistant Coq is being leveraged in the formal methods community. Coq could be a direct replacement for `Isabelle` with respect to this research, and the shown method of Haskell prototyping appears to be achievable is a similar manner with this new tool. The specifics and methodologies of this tool is not explored, but it is recommended for evaluation given its current popular and community acceptance. Using the "current popular tool" may allow for the most up-to-date implementation as to avoid issues that occurred with `Haskabelle`.

V.3 Path Forward and Recommended Actions

The last portion of the thesis looks at and proposed new paths forward based on the research herein. It is assumed that the A^2E^2 properties are acceptable and use of the Formal Verification Cycle is followed (i.e., assurance of a specification, which is followed by verification of an implementation). Assuming these two conditions, there are multiple issues that were unaddressed or identified at the conclusion of this research. This set of future work is not all-inclusive, but it aims to provide insights for any researcher following on this effort.

Security Property Theory Validation

As previously discussed, the security properties presented in Chapter 4 are currently considered as a "best practice" approach. This research was unable to exercise the proposed method of design theory assurance to individual cases of the security properties. However, once a set of tools are decided for design theory assurance, then the implementation and verification of the properties can commence. It is recommended to start with a single property to verify (e.g., Access Control). Upon the successful validation of the first property, then the remainder of the properties (or the subset of properties for a specification) could be verified by replicating the assurance process. This assurance of the security properties validates and proves the theory for the Formal Verification Life Cycle, and each successfully completion of a property allows subsequent specifications and designs to reuse the assured work. This validation could allow implicit trust when leveraging A^2E^2 properties, and potentially allow a focus on implementation efforts (as opposed to theory and property validation) in the future.

Determining Methods of Automated Assurance

Another observation has been a trend toward the use and implementation of automated formal verification. Solutions exist for "by hand" assurance techniques, but commonly these approaches are extremely rigorous and tedious. These approaches can take a considerable amount of time to construct and verify. A more preferred way is for an automation of the assurance methods for chosen security properties. Ideally, these automated tools allow a security property to be chosen or implemented into a specification with minimal effort after the first successfully use (i.e., there may need to be an investment into the first case of assurance by validation, but future work would reuse past assured property in other specifications).

An example of a tool, striving for such automation, is MIT Programming Languages & Verification Group's "Fiat" tool. The tool is stated to allow a declarative specification to generate a correct-by-construction program while providing a formal proof trail certifying that the program meets the original specification [86]. This tool could be thought analogously to the process of what `Haskabelle` does for Haskell and Isabelle, but Fiat leverages the functional programming language of OCaml and Coq [86]. Fiat's build process could shorten the time it would take to construct an implementable program, since the tool appears to automate the whole construction and process idealized in the Formal Verification Cycle. The tool was only identified recently, thus more analysis and verification of the tool and its capabilities are needed.

Native Code vs. Java-based Code Market Share Analysis

It is considered that Go language built code is equivalent, or at least in the same category of software, as native code for Android. This acknowledgment is significant, since Android typically divides it software into two categories of native code (e.g., C-based code with the Bionic library) Java-based code. There are issues of using native code applications since they execute in a different fashion than Java-based applications. For instance the example A^2E^2 Go container was a native code application, and its containment would be for other native code applications (e.g., no Java run-time environment is native to the Linux Kernel or as ELF executables, thus the only supported programs would be for native code programs). Additionally, the majority of Android programs are presumed to not be native code, so the current container would also not be able to run this majority of applications. This may stem from facts like Google's documentation recommends to not build native code applications citing potential security concerns [56]. Therefore, to ensure the greatest application coverage, with respect to A^2E^2 projects, then it should be determined what percent of applications are either native code or Java based. This information could be gained by a market share analysis of Google Play Android applications. Once this market share of applications is determined, then a final decision should be made to determine which style of Android A^2E^2 projects implement.

General Applicability of A^2E^2

An interesting side effect in the research and identification of A^2E^2 security properties was no Android-specific properties were defined. The identified security properties were abstracted in a manner such that they are not tied to a specific platform or operating system. Additionally, the implementation attempted in the research was based on Go and not any Android specifics. The tested materials and procedures were completed on personal computers, so even though the intended target was Android, the fact remains the development occurred mostly on Linux. This leaves open research directions of assured specifications and designs for general computing and on desktop operating systems.

Work Relating to Model Checking and the Formal Verification Cycle

One topic that was not explored during this research is the application of model checking for either stage of specification validation or implementation verification. It is not known what types of tools or capabilities are available for model checking, but this type of system abstraction is another popular approach in the formal methods community. It may be the case that an equivalent, or easier, approaches exist to validate A^2E^2 designs. Models for containment on Android may exist to support A^2E^2, but no research was executed thus no recommended methods or tools can be provided. This discussion seeks to present the topic and make readers aware of another avenue in assurance.

Determining Android Specific Implementation Concerns

The main effort of this research was accomplished on Linux desktop computers. Specifically, the Go compiler was leveraged to test out the mock container program, and the gomobile tool chain verified the ability to generate Android applications.

However, the mock container implementation was never ported to Android, and any specific implementation constraints and limitations for Android were never identified. An example case would be with respect to the Bionic C library. This library does not permit all native Linux API and function calls to be leveraged or executed on Android, and many of these restrictions are due to Android's security and permission model (e.g., if a phone is not rooted with full permissions and access, then typical privileged API calls would be denied). It is extremely likely that the issues seen in C and native code implementations will exist with Go-based solutions. Apiece of evidence that can support these concerns is the fact that the Go container executable must be run as root, or with sudo-based permissions, to properly execute on Linux. This is only one example, but such concerns should be considered and subsequently identified for Android execution for containment. If such restrictions were found, then it may be the case that the ability to control and implement a Go based container is impossible due to Android's current model

List of Abbreviations

Abbreviation		Page
OS	operating system	1
CVEs	Common Vulnerabilities and Exposures	1
NVD	National Vulnerabilities Database	1
NICTA	National ICT Australia	9
LLVM-IR	LLVM Intermediate Representation	12
LXC	Linux Containers	17
AuFS	Advanced Multi-layered Unification Filesystem	21
Golang	Go Programming Language	22
APK	Android Package Kit	24
SDK	Android Software Development Kit	26
UID	User IDentification	28
CIA	Confidentiality, Integrity, and Availability	28
O-ISM3	Open Information Security Management Maturity Model	28
DoD	Department of Defense	28
NIST SP	National Institute of Standards and Technology Special Publication	28
ISMSs	Information Security Management Systems	32
CFR	Code of Federal Regulations	35
FARS	Federal Acquisition Regulations Systems	35
DFARS	Defense Federal Acquisition Regulation Supplement	35
JPF	Java PathFinder	43
A^2E^2	Assured Android Execution Environments	48

Bibliography

1. J. Koetsier. (2017, May) Surprise: Google Reveals iOS Market Share Is 65% to 230% Bigger Than We Thought. Forbes Media. [Online]. Available: https://www.forbes.com/sites/johnkoetsier/2017/05/18/surprise-google-reveals-apples-ios-market-share-is-65-to-230-bigger-than-we-thought/#1fac4b335890. Retrieved on 4 October 2017.

2. B. Popper. (2017, May) Google announces over 2 billion monthly active devices on Android. Vox Media. [Online]. Available: https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users. Retrieved on 4 October 2017.

3. S. Özkan. (2017, November) Google Android : CVE security vulnerabilities, versions and detailed reports. CVEDetails.com. [Online]. Available: https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224. Retrieved on 25 November 2017.

4. S. Schmeelk. (2014, October) E6998: Formal Methods on Android. Columbia University Computer Science Department. [Online]. Available: http://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/schmidt-semantics.pdf. Retrieved on 20 December 2017.

5. E. Tran. (1999) Verification/Validation/Certification. Carnegie Mellon University. [Online]. Available: https://users.ece.cmu.edu/~koopman/des_s99/verification/. Retrieved on 2 March 2018.

6. R. Pressman, *Software Engineering: A Practitioner's Approach*, ser. McGraw-Hill higher education. Boston, 2005. [Online]. Available: https://books.google.com/books?id=bL7QZHtWvaUC.

7. M. Colins. (1998) Formal Methods. Carnegie Mellon University. [Online]. Available: https://users.ece.cmu.edu/~koopman/des_s99/formal_methods/. Retrieved on 4 October 2017.

8. Z. Tatlock. (2016) Lecture 2 - Formal Reasoning. University of Washington Computer Science & Engineering. [Online]. Available: https://courses.cs.washington.edu/courses/cse331/16wi/L02/L02-Formal-Reasoning-4up.pdf. Retrieved on 4 October 2017.

9. P. Wadler, "Monads for functional programming," in *International School on Advanced Functional Programming.* Springer, 1995, pp. 24–52.

10. What is a monad? - computerphile.

11. J. Andronick, T. Bourke, P. Derrin, K. Elphinstone, D. Greenaway, G. Klein, R. Kolanski, D. Matichuk, T. Sewell, and S. Winwood. Abstract Formal Specification of the seL4/ARMv6 API. National ICT Austrailia Limited. [Online]. Available: https://sel4.systems/Info/Docs/seL4-spec.pdf. Retrieved on 20 November 2017.

12. R. Gore. (2016) Hoare Logic. Australian National University. [Online]. Available: https://cs.anu.edu.au/courses/comp2600/Lectures/16HoareI.pdf. Retrieved on 20 December 2017.

13. S. Chong, J. Guttman, A. Datta, A. Myers, B. Pierce, P. Schaumont, T. Sherwood, and N. Zeldovich, "Report on the NSF Workshop on Formal Methods for Security," 2016.

14. R. C. Armstrong, R. J. Punnoose, M. H. Wong, and J. R. Mayo. (2014, December) Survey of Existing Tools for Formal Verification. Sandia National Laboratories. [Online]. Available: http://prod.sandia.gov/techlib/access-control.cgi/2014/1420533.pdf. Retrieved on 09 November 2017.

15. isabelle.in.tum.de. (2016, October) Overview. University of Cambridge and Technische Universitt Mnchen. [Online]. Available: http://isabelle.in.tum.de/overview.html.

16. J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

17. A. Reiter, A. Spitz, and J. Carlson. (2016, November) Leveraging Intermediate Forms for Analysis. Veracode Research and Intrepid Pursuits. [Online]. Available: https://llvm.org/devmtg/2016-11/Slides/Carlson-LeveragingIntermediateForms.pdf. Retrieved on 09 July 2017.

18. Data61. (2016) Frequently Asked Questions on seL4. Commonwealth Scientific and Industrial Research Organisation. [Online]. Available: https://wiki.sel4.systems/FrequentlyAskedQuestions. Retrieved on 18 May 2017.

19. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* ACM, 2009, pp. 207–220.

20. K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* ACM, 2013, pp. 133–150.

21. K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. (2007) Towards a Practical, Verified Kernel. [Online]. Available: https://www.usenix.org/legacy/events/hotos07/tech/full_papers/elphinstone/elphinstone_html/. Retrieved on 20 October 2017.

22. L. Mondy. (2017) Getting started with seL4, CAmkES, and L4v: Dependencies. [Online]. Available: https://research.csiro.au/tsblog/getting-started-sel4-camkes-l4v-dependencies/. Retrieved on 15 May 2017.

23. G. Klein. (2010, October) [Haskell] Specification and prover for Haskell. Haskell.org. [Online]. Available: https://mail.haskell.org/pipermail/haskell/2010-October/022348.html. Retrieved on 16 November 2017.

24. F. Haftmann, "From higher-order logic to Haskell: there and back again," in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation.* ACM, 2010, pp. 155–158.

25. G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

26. R. Shu, P. Wang, S. A. Gorski III, B. Andow, A. Nadkarni, L. Deshotels, J. Gionta, W. Enck, and X. Gu, "A Study of Security Isolation Techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, p. 50, 2016.

27. U. Steinberg and B. Kauer, "NOVA: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems.* ACM, 2010, pp. 209–222.

28. J. P. Buzen and U. O. Gagliardi, "The Evolution of Virtual Machine Architecture," in *National Computer Conference and Exposition, AFIPS 73*, 1973.

29. S. Hogg. (2014) Software Containers: Used More Frequently than Most Realize. Network World. [Online]. Available: https://www.networkworld.com/article/2226996/cisco-subnet/software-containers--used-more-frequently-than-most-realize.html. Retrieved on 4 October 2017.

30. P. Rubens. (2017, June) What are containers and why do you need them. CIO.com. [Online]. Available: http://www.cio.com/article/2924995/software/what-are-containers-and-why-do-you-need-them.html. Retrieved on 12 August 2017.

31. T. Brown, C. Wilhite, M. Keating, R. Anderson, and S. Cooley. (2016, May) Windows Containers. Microsoft. [Online]. Available: https://www.docs.microsoft.com/en-us/virtualization/windowscontainers/about. Retrieved on 4 October 2017.

32. C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. (2014, September) Use of Formal Methods at Amazon Web Services. Amazon.com. [Online]. Available: lamport.azurewebsites.net/tla/formal-methods-amazon.pdf. Retrieved on 4 October 2017.

33. Microsoft Corporation, "AWS to Azure services comparison," March 2017, retrieved on 7 June 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services.

34. B. Burns. (2017, May) Kubernetes in action: How orchestration and containers can increase uptime and resiliency. Microsoft Corporation. [Online]. Available: https://goo.gl/Tev06Z. Retrieved on 4 October 2017.

35. B. Chou. (2017, April) How were making Webflow Hosting even more reliable. Webflow, Inc. [Online]. Available: https://webflow.com/blog/how-were-making-webflow-hosting-even-more-reliable.

36. P. Abbassi. (2015, June) Reliability Is Not Enough - Building Resilient Applications With Containerized Microservices. Giant Swarm GmbH. [Online]. Available: https://blog.giantswarm.io/reliability-not-enough-resilient-applications-containerized-microservices/. Retrieved on 4 October 2017.

37. P. Ratazzi, A. Bommisetti, N. Ji, and W. Du, "PINPOINT: Efficient and Effective Resource Isolation for Mobile Security and Privacy," in *Proceedings of the SPW Workshop on Mobile Security Technologies (MoST)*, 2015.

38. W. B. Kimball, "SecureQEMU: Emulation-based Software Protection Providing Encrypted Code Execution and Page Granularity Code Signing," Air Force Institute of Technology, Tech. Rep., 2008.

39. J. Pescatore. (2017, March) Cyber Security Trends: Aiming A-head of the Target to Increase Security in 2017. SANS Institute. [Online]. Available: https://www.sans.org/reading-room/whitepapers/cloud/cyber-security-trends-aiming-target-increase-security-2017-37702. Retrieved on 4 October 2017.

40. C. Hoare, "How did software get so reliable without proof?" in *FME'96: Industrial Benefit and Advances in Formal Methods*. International Symposium of Formal Methods Europe, 1996, pp. 1–17. [Online]. Available: https://goo.gl/OFG3fc.

41. Docker Inc. (2018) Docker frequently asked questions (FAQ). [Online]. Available: https://docs.docker.com/engine/faq/. Retrieved on 11 December 2017.

42. K. Cochrane and P. Mortensen. How is Docker different from a normal virtual machine? StackOverflow.com. [Online]. Available: https://stackoverflow.com/questions/16047306/how-is-docker-different-from-a-normal-virtual-machine. Retrieved on 15 Janurary 2018.

43. S. Moser. LXC / Linux Containers. MUG.org. [Online]. Available: http://www.mug.org/wp-content/uploads/2014/01/presentation.pdf. Retrieved on 15 Janurary 2018.

44. Google Inc. (2017) Frequently Asked Questions (FAQ) - The Go Programming Language. Google. [Online]. Available: https://golang.org/doc/faq#creating_a_new_language. Retrieved on 25 October 2017.

45. P. Krill. (2017, July) Go language soars to new heights in popularity. IDG Communications, Inc. [Online]. Available: https://www.infoworld.com/article/3208904/application-development/go-language-soars-to-new-heights-in-popularity.html. Retrieved on 10 August 2017.

46. L. Dixon. (2016, October) Linux containers in 500 lines of code. [Online]. Available: https://blog.lizzie.io/linux-containers-in-500-loc.org. Retrieved on 7 June 2017.

47. L. Rice. (2016, October) Building a container from scratch in Go - Liz Rice (Microscaling Systems). Container Camp UK. [Online]. Available: https://www.youtube.com/watch?v=Utf-A4rODH8. Retrieved on 4 October 2017.

48. J. Friedman. (2016, April) Build Your Own Container Using Less than 100 Lines of Go. InfoQ.com. [Online]. Available: https://www.infoq.com/articles/build-a-container-golang. Retrieved on 4 October 2017.

49. L. Rice. (2016, August) Container from scratch. GitHub.com. [Online]. Available: https://gist.github.com/lizrice/a5ef4d175fd0cd3491c7e8d716826d27. Retrieved on 4 October 2017.

50. I. Merriam-Webster. (2017) Operating System — Definition of Operating System by Merriam-Webster. [Online]. Available: https://www.merriam-webster.com/dictionary/operating%20system. Retrieved on 4 October 2017.

51. FAUguy. (2011, August) Google's Android OS: Past, Present, and Future. phoneArena.com. [Online]. Available: https://www.phonearena.com/news/Googles-Android-OS-Past-Present-and-Future_id21273. Retrieved on 4 October 2017.

52. R. Amadeo. (2012, September) A History of Pre-Cupcake Android Codenames. Android Police. [Online]. Available: http://www.androidpolice.com/2012/09/17/a-history-of-pre-cupcake-android-codenames/. Retrieved on 4 October 2017.

53. H. Robertson. (2008, October) Android OS source code now available. Android Authority. [Online]. Available: http://www.androidauthority.com/android-os-source-code-now-available-495/. Retrieved on 4 October 2017.

54. T. Spring. (2017, September) Whats New In Android 8.0 Oreo Security. ThreatPost.com. [Online]. Available: https://threatpost.com/whats-new-in-android-8-0-oreo-security/128061/. Retrieved on 4 October 2017.

55. Google Inc. (2017, March) System and kernel security. Android. [Online]. Available: https://source.android.com/security/overview/kernel-security. Retrieved on 23 October 2017.

56. ——. (2017, April) Security Tips. Android. [Online]. Available: https://developer.android.com/training/articles/security-tips.html. Retrieved on 23 October 2017.

57. ——. (2017, April) Implementing Security. Android. [Online]. Available: https://source.android.com/security/overview/implement. Retrieved on 23 October 2017.

58. The Open Group, *Open Information Security Management Maturity Model O-ISM3, Version 2.0*, ser. The Open Group Series, September 2017.

59. United States Federal Government. Code of Federal Regulations. The United States National Archives and Records Administration. [Online]. Available: https://www.archives.gov/federal-register/cfr. Retrieved on 11 November 2017.

60. Cornell Law School. CFR - Table of Contents. Legal Information Institute. [Online]. Available: https://www.law.cornell.edu/cfr/text. Retrieved on 11 November 2017.

61. United States Federal Government. (2015, November) Part 239 - Acquisition of Information Technology. Defense Federal Acquisition Regulations. [Online]. Available: http://farsite.hill.af.mil/reghtml/regs/far2afmcfars/fardfars/dfars/dfars239.htm. Retrieved on 11 November 2017.

62. United States Departement of Commerce and National Institute of Standards and Technology. (2011, March) Information Security. NIST Special Publication 800-39. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-39.pdf. Retrieved on 11 October 2017.

63. United States Federal Government. (2016, October) 252.204-7000 Disclosure of Information. Defense Federal Acquisition Regulations. [Online]. Available: https://www.acq.osd.mil/dpap/dars/dfars/html/current/252204.htm#252.204-7012. Retrieved on 11 November 2017.

64. United States Departement of Commerce and National Institute of Standards and Technology. (2013, April) Security and Privacy Controls for Federal Information Systems and Organizations. NIST Special Publication 800-54 rev 4. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf. Retrieved on 11 October 2017.

65. United States Departement of Commerce, United States Technology Administration, and National Institute of Standards and Technology. (2004, February) Standards for Security Categorization of Federal Information and Information

Systems. Federal Information Processing Standards Publication. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.199.pdf. Retrieved on 4 October 2017.

66. A. Armando, G. Costa, and A. Merlo, "Formal Modeling and Reasoning about the Android Security Framework," *Trustworthy Global Computing*, vol. 8191, 2012.

67. M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. v. Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock Android," 2015.

68. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Java-MaC: a runtime assurance tool for Java programs," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 218–235, 2001.

69. Karlsruhe Institute of Technology, Chalmers University of Technology, and Technische Universitt Darmstadt. (2018) About – The KeY Project. The KeY Project. [Online]. Available: https://www.key-project.org/about/. Retrieved on 15 December 2017.

70. ——. (2018) Symbolic Execution Debugger (SED) – The KeY Project. The KeY Project. [Online]. Available: https://www.key-project.org/eclipse/sed/. Retrieved on 15 December 2017.

71. McGill University. (2018) Soot. Sable Research Group. [Online]. Available: https://sable.github.io/soot/. Retrieved on 30 December 2017.

72. A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 617–632, 2014.

73. GitHub Inc. (2018, January) Instrumenting Android Apps with Soot. GitHub.com. [Online]. Available: https://github.com/Sable/soot/wiki/Instrumenting-Android-Apps-with-Soot. Retrieved on 30 January 2018.

74. ——. (2016, May) How to use Soot to create call graph for Android apps? GitHub.com. [Online]. Available: https://github.com/secure-software-engineering/soot-infoflow/issues/38. Retrieved on 30 January 2018.

75. S. Anand. (2016, May) Dynamic Symbolic Execution of Android Apps. GitHub.com. [Online]. Available: https://github.com/saswatanand/acteve. Retrieved on 15 December 2017.

76. J. Jeon, K. K. Micinski, and J. S. Foster. (2012, July) SymDroid: Symbolic execution for Dalvik bytecode. [Online]. Available: www.cs.umd.edu/~jfoster/papers/cs-tr-5022.pdf. Retrieved on 27 September 2017.

77. Q. Li. SymDroid: A Symbolic Executor to Identify Activity Permission in Android Application. [Online]. Available: http://www.cs.umd.edu/grad/scholarlypapers/papers/QianwenLi.pdf. Retrieved on 27 September 2017.

78. S. Lau. (2007, October) What is JPF? [Online]. Available: https://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/what_is_jpf. Retrieved on 15 December 2017.

79. H. Botha. (2017) JPF-Android Overview. [Online]. Available: https://heila.bitbucket.io/jpf-android/. Retrieved on 30 December 2017.

80. P. Mehlitz. Open Source at Ames - PathDroid. [Online]. Available: https://ti.arc.nasa.gov/opensource/projects/pathdroid/. Retrieved on 30 December 2017.

81. J. C. Blanchette and A. Popescu. Isabelle and Security. [Online]. Available: https://people.mpi-inf.mpg.de/~jblanche/iandsec.pdf. Retrieved on 19 December 2017.

82. isabelle.in.tum.de. (2009, October) Haskabelle. University of Cambridge and Technische Universitt Mnchen. [Online]. Available: http://isabelle.in.tum.de/website-Isabelle2009/haskabelle.html. Retrieved on 4 October 2017.

83. ——. (2010, January) Haskabelle. University of Cambridge and Technische Universitt Mnchen. [Online]. Available: http://isabelle.in.tum.de/website-Isabelle2011/haskabelle.html. Retrieved on 4 October 2017.

84. C. T. Schonwald. (2015, May) github mirror of haskabelle (haskell to isabelle theoremprover tool). GitHub.com. [Online]. Available: https://github.com/cartazio/haskabelle. Retrieved on 4 October 2017.

85. T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014. [Online]. Available: https://books.google.com/books?id=lqkSswEACAAJ.

86. A. Chlipala, B. Delaware, C. Pit-Claudel, and J. Gross. Fiat - Mostly Automated Synthesis of Correct-by-Construction Programs. MIT Programming Languages & Verification Group. [Online]. Available: http://plv.csail.mit.edu/fiat/. Retrieved on 15 Janurary 2018.

87. Google Inc. (2017, March) Security Enhancements in Android 1.5 through 4.1. Android. [Online]. Available: https://source.android.com/security/enhancements/enhancements41. Retrieved on 4 October 2017.

88. ——. (2017, March) Security Enhancements in Android 4.2. Android. [Online]. Available: https://source.android.com/security/enhancements/enhancements41. Retrieved on 4 October 2017.

89. ——. (2017, March) Security Enhancements in Android 4.3. Android. [Online]. Available: https://source.android.com/security/enhancements/enhancements43. Retrieved on 4 October 2017.

90. ——. (2017, March) Security Enhancements in Android 4.4. Android. [Online]. Available: https://source.android.com/security/enhancements/enhancements44. Retrieved on 4 October 2017.

91. ——. (2017, March) Security Enhancements in Android 5.0. Android. [Online]. Available: https://source.android.com/security/enhancements/enhancements50. Retrieved on 4 October 2017.

92. ——. (2017, March) Security Enhancements in Android 6.0. Android. [Online]. Available: https://source.android.com/security/enhancements/enhancements50. Retrieved on 4 October 2017.

93. ——. (2017, March) Security Enhancements in Android 7.0. Android. [Online]. Available: https://source.android.com/security/enhancements/enhancements50. Retrieved on 4 October 2017.

Appendix A. Chapter 1 Related Documents

The following appendix contains source code and scripts leveraged in the work present in Chapter 1.

A.1 CVE Analysis - Acquire Archives Script

```
#!/bin/bash

webSite="https://static.nvd.nist.gov/feeds/json/cve/1.0/"
ZipFolder="Raw_Zips"

Zips=(
#nvdcve-1.0-modified.json.zip
#nvdcve-1.0-recent.json.zip
nvdcve-1.0-2017.json.zip
nvdcve-1.0-2016.json.zip
nvdcve-1.0-2015.json.zip
nvdcve-1.0-2014.json.zip
nvdcve-1.0-2013.json.zip
nvdcve-1.0-2012.json.zip
nvdcve-1.0-2011.json.zip
nvdcve-1.0-2010.json.zip
nvdcve-1.0-2009.json.zip
#nvdcve-1.0-2008.json.zip
#nvdcve-1.0-2007.json.zip
#nvdcve-1.0-2006.json.zip
#nvdcve-1.0-2005.json.zip
#nvdcve-1.0-2004.json.zip
#nvdcve-1.0-2003.json.zip
#nvdcve-1.0-2002.json.zip
)

if [[ ! -d "$ZipFolder" ]]; then
  mkdir -p $ZipFolder
  if [[ "$?" == "0" ]]; then
    echo "[NOTE] Created new archive folder: $ZipFolder"
  else
    echo "[ERROR] Couldn't create the archive folder!"
    exit 1
  fi
fi

pushd $ZipFolder
for zip in ${Zips[*]}; do
  if [[  ! -f "$zip" ]]; then
    echo "[NOTE] Downloading: $zip"
    wget $webSite$zip
  else
    echo "[NOTE] Skipping file ($zip). It exists!"
  fi
```

```
done
popd
```

A.2 CVE Analysis - Analysis Script

```
#!/bin/bash
ZipFolder="Raw_Zips"
ExtractFolder="Extracted_Zips"
AnalysisFolder="Analysis"
runDate=`date +"%Y-%m-%dT%H-%M"`
echo "[DEBUGGING] Run date: $runDate"

#bjson==BROKEN JSON
GlobalVulnsFile=$runDate"_GlobalList_Vulns.bjson"
echo "[DEBUGGING] Global save file: $GlobalVulnsFile"

searchString='"product_name":"android"'

zips=(`ls $ZipFolder | grep zip`)

if [[ ! -d "$ZipFolder" ]]; then
  echo "[ERROR] Zip folder DOES NOT EXIST! ($ZipFolder)"
exit
fi

if [[ ! -d "$ExtractFolder" ]]; then
  mkdir -p $ExtractFolder
  if [[ "$?" == "0" ]]; then
    echo "[NOTE] Created new unzipping folder: $ExtractFolder"
  else
    echo "[ERROR] Couldn't create the folder!"
    exit 1
  fi
fi

if [[ ! -d "$AnalysisFolder" ]]; then
  mkdir -p $AnalysisFolder
  if [[ "$?" == "0" ]]; then
      echo "[NOTE] Created new analysis folder: $AnalysisFolder"
  else
    echo "[ERROR] Couldn't create the folder!"
    exit 1
  fi
fi


pushd $ExtractFolder
  for file in ${zips[*]}; do
    echo "[NOTE] Extracting: $file"
    unzip -o ../$ZipFolder/$file
  done

  jsonSrcFiles=(‘ls *.json | grep −v ”zip”‘)
popd


pushd $AnalysisFolder
  echo ””
  echo ”[NOTE] SAVING DATA TO: $GlobalVulnsFile”
  echo ””
  echo ”###GLOBAL VULNS FILE $run” > $GlobalVulnsFile

  for srcFile in ${jsonSrcFiles[*]}; do
    echo ”[NOTE] Parsing ($searchString) from: $srcFile”
    #cat Extracted_Zips/nvdcve−1.0−2009.json | tr −d ”[:blank:]” | tr −d ”\n” | sed ”s/\”cve\”:{/\n###/g” | grep −i ”android”
    cat ../$ExtractFolder/$srcFile | tr −d ”[:blank:]” | tr −d ”\n” | sed ”s/\”cve\”:{/\n###/g” | grep −i ’”product_name”:”android”’ | grep −i ’”vendor_name”:”google”’ >> $GlobalVulnsFile
  done
popd


#Get list of CVEs matching search string
cat $GlobalVulnsFile| grep ID | grep affects | tr ”\”” ”\n” | grep ”CVE” >> $runDate”_CVEs.txt”
```

Appendix B. Chapter 2 Related Documents

The following appendix contains documentation leveraged in the work present in Chapter 2.

B.1 Android Security Features added by Version Release

The following Tables 17 and 18 are based on the documentation by Google [87] [88] [89] [90] [91] [92] [93] [54].

Table 17. Highlighted Android Security Features (1.5 to 4.4)

Android Release	Feature
1.5	Stack Protection (fstack-protector)
1.5	Integer Overflow Reduction (via safe iop and OpenBSD calloc)
1.5	Chunk Consolidation Attack Prevention (OpenBSD dlmalloc)
1.5	Format string protections (compiler format-security options)
2.3	No eXecute (NX) for code execution on the stack and heap
2.3	Mitigation of null pointer dereference escalation attacks
4.0	Address Space Layout Randomization (ASLR)
4.1	Position Independent Executable (PIE) Support
4.1	Read-only relocation/immediate binding
4.1	Ensure settings avoid leaking kernel addresses
4.2	App verification (via digital signatures)
4.2	Root privilege escalation prevention (installed)
4.2	Symlink attack prevention (0_NOFOLLOW)
4.2	ContentProvider Default Configuration for Apps
4.3	SELinux used to reinforce Android sandbox
4.3	Removed all setuid/setgid programs
4.3	Preventing applications from executing setuid programs
4.3	Capability bounding; drop unnecessary capabilities prior to execution
4.3	AndroidKeyStore Provider (restricted App private keys)
4.3	NO_NEW_PRIVS: Linux kernel version 3.5 to block new privileges prior to code execution
4.3	Detection of memory corruption vulnerabilities or unterminated string constants.
4.3	Read only relocation allowed (static linked executables); removed all text relocation
4.4	SELinux in enforcing mode for the Android Sandbox
4.4	Implemented FORTIFY_SOURCE level 2 protections

Table 18. **Highlighted Android Security Features (5.0 to 8.0)**

Android Release	**Feature**
5.0	Full disk encryption by default
5.0	Requirement of all dynamically linked executables to support PIE
5.0	SELinux enforcing mode is required for all domains
5.0	non-PIE linker support removed
5.0	Improvements to FORTIFY_SOURCE
6.0	Applications request permissions at run time (vs. install time)
6.0	Hardware-Isolated Security via new HAL to protect Kernel/local access compromise
6.0	SELinux enforced polices to ensure better isolation of users, /proc access, etc.
6.0	File-based encryption (vs. a single storage area)
7.0	SELinux enhancements for application sandbox
7.0	Kernel hardening (read only portions, user space addresses, etc.)
8.0	"Project Treble" - isolation of patching/Android processes from vendor-specific changes
8.0	Migration from ASLR, format string, and fstack-protector
8.0	Kernel protections with seccomp filtering
8.0	Per app approval of "unknown sources" installation

Appendix C. Chapter 4 Related Documents

The following appendix contains source code and scripts leveraged in the work present in Chapter 4.

C.1 KLEE: KLEE Docker Install Script

```
#!/bin/bash
#From: https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/#install-docker-ce

echo "[KDI] Starting KLEE Docker Installer"
echo "[KDI] =============================="

echo "[KDI] Cleaning up prior Docker Installation..."
sudo apt-get update

sudo apt-get remove -y docker docker-engine docker.io

#FOR Ubuntu 14.04:
#sudo apt-get install \
#     linux-image-extra-$(uname -r) \
#     linux-image-extra-virtual

echo "[KDI] Installing Deps (apt-transport-https ca-certs curl sw-props-common)..."
sudo apt-get install -y\
     apt-transport-https \
     ca-certificates \
     curl \
     software-properties-common

echo "[KDI] Installing GPG Key..."
InstalledKey=`curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add - 2>&1 | grep "OK" | wc -l`

if [[ "$InstalledKey" != "1" ]]; then
  echo "[KDI] [ERROR] The Ubuntu GPG Key DID NOT install!"
  exit
fi

VerifyKey=`sudo apt-key fingerprint 0EBFCD88 2>&1 | grep "rsa4096 2017-02-22" | wc -l`
if [[ "$VerifyKey" != "2" ]]; then
  echo "[KDI] [ERROR] The Ubuntu GPG Key Has CHANGED!"

  #Two lines return from this expected result:
    #pub   rsa4096 2017-02-22 [SCEA]
    #      9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
    #uid           [ unknown] Docker Release (CE deb) <docker@docker.com>
    #sub    rsa4096 2017-02-22 [S]
  exit
fi

echo "[KDI] Setting up the docker stable release repo..."
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
sudo apt-get update

echo "[KDI] Installing Docker..."
sudo apt-get install -y docker-ce

echo "[KDI] Running Docker Hello World..."
sudo docker run hello-world

#echo "[KDI] Grabbing KLEE pre-built"
#The instructions state that there is 3rd party code...so I'm building it for now...
#docker pull klee/klee

echo "[KDI] Grabbing KLEE-needed Docker files..."
git clone https://github.com/klee/klee.git
cd klee

sudo usermod -aG docker $USER

docker build -t klee/klee .
docker run --rm -ti --ulimit='stack=-1:-1' klee/klee

whoami
```

C.2 KLEE: KLEE Docker Start Script

```
#!/bin/bash
#This is NOT persistent: docker run --rm -ti --ulimit='stack=-1:-1' klee/klee

containerName="my_first_klee_container"
containerMade=`docker ps -a | grep my_first_klee_container | wc -l`

echo "[KCL] Starting KLEE Container Launcher"
echo "[KCL] =================================="

echo "[DEBUGGING] containerMade val: $containerMade"

if [[ "$containerMade" == "1" ]]; then
  echo "[KCL] Container FOUND! Starting old instance..."
  docker start -ai $containerName
else
  echo "[KCL] Container NOT FOUND! Creating NEW instance..."
  docker run -ti --name=$containerName --ulimit='stack=-1:-1' klee/klee
fi
```

C.3 GoLLVM Ubuntu Installer Script

```
#!/bin/bash
#// Here 'workarea' will contain a copy of the LLVM source tree and one or more build areas

echo "[SGU] Starting Setup Gollvm for Ubuntu"
echo "[SGU] ================================"

echo "[SGU] Installing Deps (gmp mpfr mpc)"
#sudo apt-get install -y git cmake m4 build-essential libgmp-dev libmpfr-dev libmpc-dev libxml2-dev ocaml libctypes-ocaml
sudo apt-get update
sudo apt-get install -y git cmake m4 build-essential libgmp-dev libmpfr-dev libmpc-dev gccgo-6

#Get Ninja
#https://github.com/ninja-build/ninja/releases
ninjaInstalled=`which ninja | wc -l`
if [[ $ninjaInstalled == "0" ]]; then
echo "[SGU] Installing Ninja"
  wget https://github.com/ninja-build/ninja/releases/download/v1.8.2/ninja-linux.zip
  if [[ -f ninja-linux.zip ]]; then
    unzip ninja-linux.zip
    sudo mv ninja /usr/bin/
  fi
fi

ninjaInstalled=`which ninja | wc -l`
if [[ $ninjaInstalled == "1" ]]; then
echo "[SGU] Ninja installed"
echo "[SGU] Cloning LLVM"
  #Sources
  git clone http://llvm.org/git/llvm.git

  if [[ -d llvm ]]; then
    pushd llvm > /dev/null
    git reset --hard 3962d561a63fb3912c9310838793863ce5818cba #Last commit from Than McIntosh, pre-LLVM 3 changes (Sep 27, 2017)
    cd tools

    echo "[SGU] Cloning Clang"
    git clone http://llvm.org/git/clang.git
    if [[ -d clang ]]; then
      pushd clang
      git reset --hard 29487927c0f5d8cd6b23978a0216b17041161cc5 #Last commit on Sep 27, 2017
      #OPTIONAL TOOLS:
```

```
      cd tools
      git clone http://llvm.org/git/clang-tools-extra.git extra
      cd extra
      git reset --hard 1c6297911930ae6ad88a3383eb4c88a27460eb54 #Last commit on Sep 27, 2017
      popd
    else
      echo "[SGU] [ERROR] Clang clone FAILED!"
    fi

    echo "[SGU] Cloning gollvm"
    git clone https://go.googlesource.com/gollvm
    if [[ -d gollvm ]]; then
      cd gollvm/
      git reset --hard 0b6e1072828dd59cead801c01d548675bedae644 #Most recent of this script creation
      cd llvm-gofrontend
      git clone https://go.googlesource.com/gofrontend
      cd gofrontend
      git reset --hard adc6eb826f156d0980f0ad9f9efc5c919ec4905e #Most recent of this script creation
      #cd ../../../..
      popd > /dev/null

      #// Create a build directory and run cmake
      echo "[SGU] Building gollvm"
      mkdir -p build.opt
      cd build.opt
      cmake -DCMAKE_BUILD_TYPE=Debug -G Ninja ../llvm #Clang dev reccomends: -DLLVM_BUILD_TESTS=ON # Enable tests; default is off.

      #// Prebuild
      echo "[SGU] Starting ninja build of gmp/mpfr/mpc"
      ninja libgmp libmpfr libmpc

      #// Now regular build
      #ninja <gollvm target(s)>
      echo "[SGU] Starting ninja build of LLVM/Clang/gollvm "
      #JUST FOR gollvm: ninja llvm-goparse
      #ninja all
      ninja install
      echo "[SGU] DONE!"
    else
      echo "[SGU] [ERROR] gollvm clone FAILED!"
    fi

  else
    echo "[SGU] [ERROR] LLVM clone FAILED!"
  fi

else
```

```
echo "[SGU] [ERROR] Ninja NOT INSTALLED!"
fi
```

C.4 GoLLVM Example Go Code

```
//A simple example of Go to test with gollvm: llvm-goparse -dump-ir -o Example.IR example.go
package foo

func main() int {
  return 1
}
```

C.5 GoLLVM Attempted KLEE Integration Code

```
package foo

//import (
//  "unsafe"
//)

func main() int{
  var a int
//  klee_make_symbolic(&a,unsafe.Sizeof(a),1)
  return get_sign(a)
}

func get_sign(x int) int {
  if x == 0 {
    return 0
  }
  if x < 0 {
    return -1
  } else {
    return 1
  }
}
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)* 22–03–2018	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2016 — Mar 2018

4. TITLE AND SUBTITLE

Assured Android Execution Environments

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Froberg, Brandon P., Capt, USAF

5d. PROJECT NUMBER

N/A

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-18-M-027

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory - Information Directorate
Information Exploitation & Operations Division
525 Brooks Road
Rome, NY 13441-4505
DSN 578-4459, COMM 315-330-4459
Email: edward.ratazzi@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RIG

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

This material is declared a workd of the U.S. Government and is not subject to copyright protection in the United States

14. ABSTRACT

The goal of this research is to determine methods of assuring isolation when executing Android software in a contained environment. A three-stage methodology called "The Formal Verification Cycle" is presented. This cycle focuses on the iteration over a set of security properties to validate each within a specification and their verification within a software implementation. A security property can be validated when its functional language prototype (e.g. a Haskell coded version of the property) is converted and processed by a formal method (e.g. a theorem proof assistant). This validation of the property enables the definition of the property in a software specification, which can be implemented separately in an imperative programming language (e.g. the Go programming language). Once the implementation is complete another formal method can be used (e.g symbolic execution) to verify the imperative implementation satisfies the validated specification. Successful completion of this cycle shows a given implmentation is equivalent to a functional language prototype, and this cycle assures a specification for the original desired security properties was properly implemented.

15. SUBJECT TERMS

Formal Methods, Android, Isabelle, KLEE, containerization, containment, containers, abstract intrepretation, symbolic execution, theorem proving, modeling

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Laurence D. Merkle, AFIT/ENG
U	U	U	U	126	**19b. TELEPHONE NUMBER** *(include area code)* (937)255-3636 x4526; Laurence.Merkle@afit.edu